ECE653
Software Testing, Quality Assurance, and Maintenance
Assignment 2 (80 Points), Version 1

Instructor: Werner Dietl
Release Date: June 4, 2019

Due: 21:00, Friday, July 5, 2019
Submit: An electronic copy on GitHub

The GitHub repository with the source code and test cases for the assignment can be obtained using `https://classroom.github.com/a/TODO`.

An account on `eceUbuntu.uwaterloo.ca` is available to you. Several of the resources required for this assignment might be already installed on these servers, but can also be downloaded independently. If you are attempting to connect to a server from off campus, remember you will need to connect to the University's VPN first: `https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn/about-virtual-private-network-vpn`

You will need a working Python environment to complete the assignment. Follow the instructions on the course web page to set one up on `eceUbuntu` or on your personal machine.

You can also use a provided Docker container. Instructions on how to install it are available at `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/07/docker-tutorial`. We will expect your assignments to work in the container.

I expect each of you to do the assignment independently. I will follow UW's Policy 71 for all cases of plagiarism.

Submission Instructions:

Please read the following instructions carefully. **If you do not follow the instructions, you may be penalized up to 5 points.** Illegible answers receive no points.

Submit by pushing your changes to the `master` branch of your GitHub repository. The repository must contain the following:

- a `user.yml` file with your UWaterloo user information;
- a single pdf file called `a2_sub.pdf`. The first page must include your full name, 8-digit student number and your uwaterloo email address;
- a directory `a2q3` that includes your code for Question 3; and
- a directory `wlang` that includes your code for Question 4.

Question 1 (10 points)

(based on a question by Marsha Chechik)

Consider the following program `Prog1`:

```
1  havoc x, y;
2  if x + y > 10 then {
3    x = x + 1;
4    y = y - 2 }
5  else {
6    y = y + 7;
7    x = x - 3 };
8
9  x = x + 2;
10
11 if 2 * (x + y) > 27 then {
12   x = x * 3;
13   y = y * 2 }
14 else {
15   x = x * 4;
16   y = y * 3 + x };
17 skip
```

(a) How many execution paths does `Prog1` have? List all the paths as a sequence of line numbers taken on the path.

(b) Symbolically execute each path and provide the resulting path condition. Show the steps of symbolic execution as a table. An example of executing the first line is given below:

Edge	**Symbolic State (PV)**	**Path Condition (PC)**
$1 \rightarrow 2$	$\mathrm{x} \mapsto X_0, y \mapsto Y_0$	true
...	...	...

(c) For each path in part (b), indicate whether it is feasible or not. For each feasible path, give values for X_0 and Y_0 that satisfy the path condition.

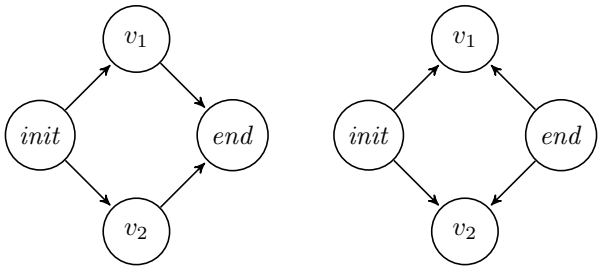

Figure 1: Two graphs.

Question 2 (10 points)

(a) The constraint *at-most-one*$(a_1, \ldots, a_n)$ is satisfied if at most one of the Boolean variables $a_1, \ldots, a_n$ is true. For example, *at-most-one*$(\top, \bot, \bot)$ is true, and *at-most-one*$(\top, \bot, \top)$ is false. Encode the constraint

$$\textit{at-most-one}(a_1, a_2, a_3, a_4)$$

into an equivalent set of clauses (i.e., in CNF).

(b) Let $G = (V, E)$ be a directed acyclic graph (DAG) with vertices V and edges $E \subseteq V \times V$, and two designated vertices $v_{init}, v_{end} \in V$. *Graph Reachability* is a decision problem that is true whenever there is a path in G that connects vertex v_{init} with v_{end}.

In this question, you have to reduce Graph Reachability to Propositional Satisfiability. Specifically, you have to develop a set of clauses in CNF, denoted *Reachable*$(G, V, v_{init}, v_{end})$, such that the clauses *Reachable* are satisfiable if and only if there is a path from v_{init} to v_{end} in G. Your reduction must be polynomial in the size of the graph G.

For example, *Reachable* is satisfiable for the graph in the left part of Figure 1, and is unsatisfiable for the graph in the right part of Figure 1.

Explain how many propositional variables you need, what do they denote, and show that your constraint is correct.

(c) *(Bonus question)* Extend your encoding from part (a) to n variables and use at most $O(n)$ clauses and variables. If your solution is based on external resources, make sure to properly reference them.

Question 3 (15 points)

In recreational mathematics, a **magic square** is a $n \times n$ square grid filled with distinct positive integers from 1 to n^2 inclusive such that each cell contains a different integer, and the sum of the integers in each row, column, and diagonal is equal.

The following is an example of 3×3 magic square:

8	1	6
3	5	7
4	9	2

(a) Write down quantifier free constraints in First Order Logic to solve the puzzle above for any positive integer n.

(b) Use Z3 Python API to implement a solver for magic square puzzles. Your solver should accept four parameters: n, r, c, val, where:

- n is the size of the puzzle – number of cells on each side, $n > 0$
- r is a row number, $0 \leq r < n$
- c is a column number, $0 \leq c < n$
- val is an integer value, $1 \leq val \leq n^2$

Your program should find a magic square of the given size n with the value val filled at location (r, c), assuming the top left corner corresponds to $(0, 0)$.

Your solver should return a 2D array of integers corresponding to the solution. If there is more than one solution, just return any one of them. If there is no such magic square, your solver should return `None`. For example,

```
>>> res = solve_magic_square(3, 1, 1, 5)
>>> print_square(res, 3)
4       9       2
3       5       7
8       1       6
```

The skeleton for the solver is provided in `a2q3/magic_square.py`.

You might find it helpful to use Z3 `z3.Distinct(x)` to create a constraint that states that all constants in the list `x` have distinct values. For example,

```
>>> x, y = z3.Ints ('x y')
>>> z3.Distinct (x, y)
x != y
```

(c) Extend the test suite in `a2q3/puzzle_tests.py` with two additional set of parameters, and one extra set of parameters that does not have a solution (i.e., the solver returns `None`).

Recall that you can execute the test suite using the following command:

```
python -m a2q3.test
```

Question 4 (30 points)

Your GitHub repository includes an implementation of a parser and interpreter for the WHILE language from the lecture notes. Your task is to write a symbolic execution engine for it.

The implementation of the interpreter is located in directory `wlang`. You can execute the interpreter using the following command:

```
(venv) $ python -m wlang.int wlang/test1.prg
x: 10
```

A sample program is provided for your convinience in `wlang/test1.prg`

You can execute the interpreter using the following command:

```
(venv) $ python -m wlang.sym wlang/test1.prg
```

A skeleton for a symbolic interpreter is given in `wlang/sym.py`. It includes an implementation of a symbolic state in a class `SymState`. The class is provided for your convenience. You are free to modify it in any way or create your own.

You may find it helpful to look at the implementation of the concrete interpreter in `wlang/int.py`. Note that the concrete interpreter takes a `State` as input, and returns a single `State` as output. On the other hand, symbolic interpreter takes a symbolic state `State` as input, and returns a **list** of symbolic states as output, where each output state corresponds to some execution of the program. In general, the number of output states will be proportional to the number of execution paths in the program.

(a) Implement symbolic execution of straight line code (i.e., programs without `if`- and `while`-statements);

(b) Extend your answer to symbolic execution of programs with `if`-statements;

(c) Extend your answer to symbolic execution of programs with `while`-statements. To handle arbitrary loops, assume that the loop is executed at most **10** times. That is, your symbolic execution engine should explore all feasible program paths in which the body of each loop is executed no more than **10** times.

(d) Extend the test suite `test_sym.py` to achieve 100% branch coverage of your implementation in parts (a), (b), and (c). Recall that you can run the test suite using

```
(venv) $ python -m wlang.test
```

and measure coverage of the test suite using

```
(venv) $ coverage run -m wlang.test
(venv) $ coverage html
```

(e) Provide a program on which your symbolic execution engine diverges (i.e., takes longer than a few seconds to run).

Question 5 (15 points)

(a) Show whether the following First Order Logic (FOL) sentence is valid or not. Either give a proof of validity, or show a model in which the sentence is false.

$$(\forall x \cdot \exists y \cdot P(x) \vee Q(y)) \iff (\forall x \cdot P(x)) \vee (\exists y \cdot Q(y))$$

(b) Show whether the following First Order Logic (FOL) sentence is valid or not. Either give a proof of validity, or show a model in which the sentence is false.

$$(\forall x \cdot \exists y \cdot P(x,y) \vee Q(x,y)) \implies (\forall x \cdot \exists y \cdot P(x,y)) \vee (\forall x \cdot \exists y \cdot Q(x,y))$$

(c) Consider the following FOL formula Φ:

$$\exists x \exists y \exists z \, (P(x,y) \wedge P(z,y) \wedge P(x,z) \wedge \neg P(z,x))$$

For each of the following FOL models, explain whether they satisfy or violate the formula Φ.

(a) $M_1 = \langle S_1, P_1 \rangle$, where $S_1 = \mathbb{N}$, and $P_1 = \{(x,y) \mid x,y \in \mathbb{N} \wedge x < y\}$. Does $M_1 \models \Phi$?

(b) $M_2 = \langle S_2, P_2 \rangle$, where $S_2 = \mathbb{N}$ and $P_2 = \{(x, x+1) \mid x \in \mathbb{N}\}$. Does $M_2 \models \Phi$?

(c) $M_3 = \langle S_3, P_3 \rangle$, where $S_3 = \mathcal{P}(\mathbb{N})$, the powerset of natural numbers, and $P_3 = \{(A,B) \mid A,B \subseteq \mathbb{N} \wedge A \subseteq B\}$. Does $M_3 \models \Phi$?

(d) Express in FOL: "Location i is a pivot of an array A such that all elements in locations lower than i are less than any elements in locations higher than i". You can use, without defining, all predicates and functions of arithmetic together with the following constants, functions, and predicates, and assume that they have the standard interpretation:

$\text{isArray}_{/1}$	true if the argument is an array
$0_{/0}, 1_{/0}, \ldots$	integer constants
$A_{/0}, B_{/0}$	constants of array sort/type
$=_{/2}, <_{/2}, \leq_{/2}$	equality and comparison
$\text{len}_{/1}$	size of an array, e.g., $\text{len}(A)$
$\text{read}_{/2}$	value of an array element at a given position, e.g., $\text{read}(A,4)$

A constant B is an array iff it satisfies $\text{isArray}(B)$. In this case, the locations of B start at 0 and go up to (but not including) $len(B)$.

(e) Express in FOL: *an array A is a permutation of an array B.* That is, both A and B have exactly the same elements but possibly ordered differently. You can use the same assumptions as in part (c).

(f) A *stack* is a well-known abstract data-structure. It can be formalized by a special constant *nil* that represents an empty stack, a binary function $push(x,y)$ that returns a new stack that is a result of pushing a value y on top of stack x, a unary function $pop(x)$ that returns a new stack obtained by removing top element from stack x, a unary function $top(x)$ that returns the top element of the stack, and a unary function $empty(x)$ that returns true iff x is an empty stack.

Axiomatize these operations in FOL with equality by writing a set of FOL formulas, denoted $STACK$, such that any model M of $STACK$ corresponds to a single abstract stack. That is, if $M \models STACK$ then M is a stack.

Hint 1: One of the formula in $STACK$ might be:

$$\forall x, y \cdot top(push(x,y)) = y$$

Hint 2: Your solution may use helper functions, e.g., $isOnStack(x)$.