Software Analysis and Verification - Fall 2020

Ruzica Piskac

December 1, 2020

You will be given details about the project submission / evaluation in a separate email.

The deadline for Project 04 is December 18th, 5:00PM.

You are encouraged to work with a partner. In this case, only one of you should submit the project, while indicating who you worked with.

It is expected that you will be able to reuse code from Project 1 in completing this project. If you previously worked with a partner, but are not working with a partner now, you may both still reuse any code from your original project. If you are working with a different partner than you worked with on Project 1, you may reuse code from either project.

1 Symbolic Execution Engine

Build a symbolic execution engine (SEE) based on a forward execution of the program.

Your SEE should treat the inputs to the program (as specified by the variables after the name of the program) as symbolic values. To check the resulting path constraints and assertions, connect your engine to a theorem prover. You should use SMT-LIB format and an SMT solver of your choice.

As always, you are welcome to use any language of your choice. Bill does need to be able to run your project, so if you want to use something obscure, check with Bill first.

Check `lang.txt`, on Canvas, to understand the syntax used to define the programs. This syntax is intentional similar to the syntax for Project 1, so we expect you will be able to reuse pieces of your code from that project. Independently of your choice of the language to implement the SEE, you need to use the program syntax given in this file.

2 Input

On the command line, your tool will be passed (as anonymous arguments) a file name, and a non negative integer n. The program should be parsed from the provided file. During symbolic execution, all execution paths through the program should be considered, up to unrolling each loop at least n times. If $n = 0$, then the loop condition is assumed to be initially false. If $n = 1$, two paths should be explored: one where the loop condition is assumed to be false, and one where the loop body is executed exactly once.

3 Output

Your tool should output the string "No violations found" if no assertion violations are found. If there are assertion violation, output each assertion violation on a separate line, as the program name, followed by a white space separated list of integers corresponding to input values that violate the assertion. Your output should not contain any other characters, EXCEPT it may contain arbitrary whitespace. If you do not follow these instructions, you will be asked to resubmit.

4 Example

Program:

```
program simple(x y)
is
  while x > y do
    x := x - 1;
  end
  assert x != y;
end
```

Call (assuming the program is invoked with ./see, and the program is in simple.imp):

```
./see simple.imp 2
```

Output

```
simple 1 1
simple 1 0
simple 2 0
```

Notice that we get three assertion violations: one resulting from the loop condition initially being false, one resulting from unrolling the loop once, and one resulting from unrolling the loop twice.

5 Other information

It is possible that we issue some amendments to the language or that I add extra material. If so, I will signal it in a Canvas announcement.

Additionally, keep the benchmarks that you use to test your code. You should submit five (or more) benchmarks by alongside your project, so that we can prepare them for the competition. Your benchmarks should include at least five programs in the IMP language with loops and annotations. Please try to write programs that do something interesting i.e. sort an array, as opposed to programs that are meaningless/artificially constructed. These benchmarks may be modified versions of benchmarks you submitted for project 1.

Notes on the language:

1. Double assignment to arrays is not supported. (i.e., you cannot write a[0], a[1] = a[1], a[0])