COMPSCI 630 Systems **Spring 2018**

Last Lecture: Testing and Program Verification

Lecturer: Emery Berger *Scribe(s): Amit Rawat*

Program Verification

"Proves" the absence of bugs. Testing on the other hand is best described by Dijkstra's quote: "proves the presence of bugs but not their absence".

THe idea is to prove that the specification for a program and its implementation produce the same output for all possible inputs. This is called complete program verification.

For example, the specification of sorted is

$$sorted(l) = \forall i \in 0, len(l) - 1 : l[i] \leq l[i+1]$$

. A sort routine then has an empty precondition and maintains the postcondition that the output list conforms to sorted. Such a triple is also called a Hoare triple.

Partial program verification on the other hand only proves the absence of some of the bugs, for e.g. buffer overflow. Type systems usually prove some properties about a program, and can be viewed as a partial program verification.

Constraints can be generated from a program which are then fed to a SAT solver to prove that some property holds true. In case the SAT solver cannot prove that the property is true, it could be either that the program is incorrect, the specification is incorrect, or even worse both are incorrect. SAT solvers try to solve a NP-complete problem namely SAT, so a polynomial execution time is not guaranteed. However, they do perform very well on most practical inputs.

Eiffel language has precond and postcond that can be added to a function, which are basically assertions that are made at runtime. The problem with runtime verification is that it increases the execution time and is input dependant, and thus only tells us about the current/past execution and not the future.

Our goal is to write correct working software. Specification for a program can be written not only for correctness, but also for bounded execution time, security etc.

Specification problem

The problem with writing a specification for a program is that writing a specification is hard and could be 3-5x the length of the program itself! This then leads to the questions of whether the specification in itself correct or not.

Rather than writing the whole specification for the program, generally a subset for critical components is written and verified against the implementation.

Testing

"I have only proved it correct, I have not tried it." - Knuth

Many forms of testing like systems and end-to-end testing. The general idea is to test the output of the program for inputs for which correct output is known. Once an input is known for which the program provides incorrect output, the goal is to "fix" the program to correct it. In case the input is very large, it can be trimmed by doing a binary search over it to find a very small input for which the program fails to run correctly.

Tests are usually written by humans, and usually have the following characteristics:

- - correlated with the code,
- - have low coverge usually,
- + test created post hoc can be good insurance for future,
- + help in documenting how the code should run,

Fuzz testing

Provide random inputs to a program to make it crash. The issue with this approach is that it has almost negligible property that the fuzzer can create a useful test case which can actually find bugs in a program. Better techniques like "concolic testing" which is a combination of concrete and symbolic execution are known to make fuzzing better. AFL (American Fuzzy Lop) is one of the framework which does intelligent fuzz testing.