Software Verification for Developers

Dr. Yegor Derevenets
yegor.derevenets@gmail.com

Berlin
15.01.2020

Software Verification

Definition

Software verification is the process of determination whether the software is correct, i.e., meets the specification.

Corollary

You cannot talk about correctness if you do not have a specification.

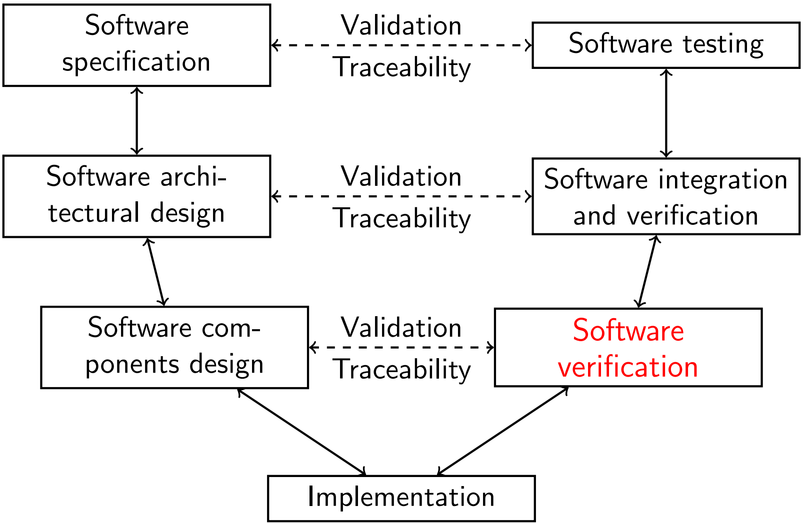
Corollary

Even if your program is correct, it does not mean that it does what you actually want, because the specification can be wrong.

Example

Boeing 737 MAX's MCAS software was built to the specification, but contributed to two plane crashes.

V-Model


Software specification
Validation
Traceability
Software testing
Software architectural design
Validation
Traceability
Software integration and verification
Software components design
Validation
Traceability
Software verification
Implementation

Software Verification Methods

Methods for ensuring correctness:

- formal verification by theorem proving,
- model checking (for finite-state programs).

Methods for finding bugs:

- model checking,
- testing,
- concolic testing,
- fuzzing,
- testing with instrumentation (e.g., sanitizers).

Definition and Motivation

- Software Verification
- V-Model

Software Verification Methods

- Formal Verification by Theorem Proving
- Model Checking
- Testing
- Concolic Testing
- Fuzzing
- Sanitizers

Summary

Formal Verification by Theorem Proving

Workflow:

1. Annotate your source code with a formal specification.
2. Generate a set of theorems from the annotated source code.
3. Prove the theorems.

Outcomes:

- All theorems are proven $\Longrightarrow$ your program is correct.
- Unable to prove $\Longrightarrow$ bug in the program *or* in the spec.

Tools:

- Frama-C/Why (CEA, INRIA, free software),
- VCC (Microsoft, non-free).

Theorem provers:

- automatic: Alt-Ergo, CVC4, Simplify, Yices, Z3...
- proof assistants: Coq, PVS, Isabelle/HOL.

Formal Verification by Theorem Proving: Example

```
void qs(int *s_arr, int first, int last) {
    if (first < last) {
        int left = first, right = last,
            middle = s_arr[(left + right) / 2];
        do {
            while (s_arr[left] < middle) left++;
            while (s_arr[right] > middle) right--;
            if (left <= right) {
                int tmp = s_arr[left];
                s_arr[left] = s_arr[right];
                s_arr[right] = tmp;
                left++; right--;
            }
        } while (left <= right);
        qs(s_arr, first, right);
        qs(s_arr, left, last);
    }
}
```

Model Checking

Workflow:

- Define a model of your program.
- Specify bad states.
- Run a model checker.

Outcomes:

- No reachable bad states found $\Longrightarrow$ the *model* is correct.
- A reachable bad state found $\Longrightarrow$ you have found a bug.
- Model checker timed out $\Longrightarrow$ unclear, too large state space.

Model checkers:

- SPIN (Promela, LTL),
- Java Pathfinder (JVM-based),
- DIVINE (LLVM-based).

Testing

Workflow:

1. Specify bad states using assertions.
2. Define test inputs.
3. Run the program with the inputs.

Outcomes:

- An assertion has failed $\Longrightarrow$ you have found a bug.
- No assertion has failed $\not\Longrightarrow$ correctness.

Tools:

- unit-test libraries: Google Test, pytest, myriads of them,
- your favourite `<assert.h>`.

Offensive programming: turn your program into one big test!

- Check everything that you can: function arguments, invariants, system calls, things that can never happen.
- Crash whenever you see something unexpected.

Code Coverage: How good are your tests?

Lots of different kinds:

- function coverage,
- statement coverage,
- branch coverage,
- condition coverage. . .

Tools: your favourite compiler.

Ways to increase coverage (coming next):

- concolic testing,
- fuzzing.

Concolic Testing 1/2

Idea: Generate test inputs exploring all code paths automatically using a SMT solver.

```
void handle_request(const char *s) {
  L1: if (s[0] == 'G') {
    L2: if (s[1] == 'E') {
      L3: if (s[2] == 'T') {
        L4:
      } else {
        L5: abort();
} } } }
```

- L1 $\rightarrow$ L2 : $s[0] =$ 'G' $\rightarrow s =$ 'G'
- L2 $\rightarrow$ L3 : $s[0] =$ 'G' $\wedge\ s[1] =$ 'E' $\rightarrow s =$ 'GE'
- L3 $\rightarrow$ L4 : $s[0] =$ 'G' $\wedge\ s[1] =$ 'E' $\wedge\ s[2] =$ 'T' $\rightarrow s =$ 'GET'
- L3 $\rightarrow$ L5 : $s[0] =$ 'G' $\wedge\ s[1] =$ 'E' $\wedge\ s[2] \neq$ 'T' $\rightarrow s =$ 'GEZ'

Concolic Testing 2/2

Tools:

- KLEE (LLVM-based, Stanford, free software),
- jCUTE (Java, Urbana-Champaign, non-free)...

There are more tools for Java, JavaScript, .NET, Erlang, and even native code!

`https://en.wikipedia.org/wiki/Concolic_testing#Tools`

Fuzzing

Idea: Generate test inputs exploring all code paths automatically using a random number generator.

Tool — libFuzzer from LLVM:

1. Write a test function accepting a buffer with the test input.
2. Compile the test with `clang -fsanitize=fuzzer`.
3. Run the test and get test inputs generated.

Sanitizers

Idea: Let the compiler instrument your program with code detecting undefined behavior at run time.

Sanitizers in Clang and GCC:

- Address Sanitizer: buffer overflows, use after free, memory leaks.
- Thread Sanitizer: data races, potential deadlocks.
- Undefined Behavior Sanitizer: signed integer overflow, float overflow, division by zero, invalid enum values...
- Memory Sanitizer: use of uninitialized memory.

Workflow:

1. Compile your program with `-fsanitize={a,t,ub,m}san`.
2. Run your program and get errors reported to stderr.

Sanitizers + Fuzzing: Find inputs leading to undefined behavior.

Summary 1/3

Software Verification Using Theorem Proving:

+ Can be used to ensure correctness.
– Requires special training and skills.
– Proofs with proof assistants are time-consuming.
– Automatic provers might not take you far enough.

Model Checking:

+ Can be used to ensure correctness.
+ Easy to learn and use.
+ Great for verifying protocols, data structures, parallel algorithms (SPIN!).
– What you check is not what you execute.

Summary 2/3

Testing:

- − Cannot be used for ensuring correctness.
- \+ Can be used to find bugs.
- \+ Does not require special training or skills.
- \+ Quality of tests can be estimated using code coverage tools.

Concolic Testing:

- \+ Can be used to increase coverage.
- − Tools require some work to get running.
- − Performance depends on SMT solver, mileage may vary.

Fuzzing:

- \+ Can be used to increase coverage.
- \+ Clang comes with fuzzing built in.

Summary 3/3

Sanitizers:

+ Require little work to get running.
+ Require little learning: just read the output and fix the problems.
+ Detect lots of programming errors that are hard/impossible to detect with other tools.
– Slowdown 2x–20x, increased memory consumption.
– Ideally all code (including dependencies) should be recompiled with sanitizers. MSan (detector of uninitialized reads) actually requires this!

Questions?

yegor.derevenets@gmail.com