Towards Reasoning of Program Logic

19CSE205 : PROGRAM REASONING

Dr. Swaminathan J

Assistant Professor

Department of Computer Science and Engineering

AMRITA VISHWA VIDYAPEETHAM

Jul - Dec 2020

Contents

Correctness of program logic implies realization of program's goal.

- We noted that ensuring lexical-syntax-semantic correctness are necessary but not sufficient to achieve program's objective.

Realization of program's objective requires at least two things.

1. A way to specify the objective.
   - Simple yet powerful
2. A means to verify if the objective is met.
   - Minimize human intervention

For terminating programs, a way to do that is to specify the expected output. But, output depends on input. Hence, specify input-output relation. Broadly, there have been two approaches.

1. Testing
2. Proofs

1. Testing

in^1 in^2 in^3 in^4 — Inputs

$\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$

Program — Execute

$\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$

out^1 out^2 out^3 out^4 — Actual output

$\Updownarrow$ $\Updownarrow$ $\Updownarrow$ $\Updownarrow$

exp^1 exp^2 exp^3 exp^4 — Expected output

✓ X ✓ ✓

Characteristics of testing

- Enumerate input-expected output pairs (test cases)
- Check conformance by execution
- Dynamic technique
- Black-box based
- Accurate
- Incomplete

De-facto and widely adopted

Dijkstra's famous quote: Testing can only prove the presence of errors but hopelessly inadequate to prove their absence.

How do we know the test cases cover all paths?

Program

```
...
if (x > 0) {
    1A
}
else {
    1B
}
...
if (y > 0) {
    2A
}
else {
    2B
}
...
```

Potentially four execution paths

$\Rightarrow$

1. ... 1A ... 2A ...
2. ... 1A ... 2B ...
3. ... 1B ... 2A ...
4. ... 1B ... 2B ...

Test cases should cover all four paths

Corresponding to branching choices

$\leftrightarrow (x > 0, y > 0)$
$\leftrightarrow (x > 0, y \leq 0)$
$\leftrightarrow (x \leq 0, y > 0)$
$\leftrightarrow (x \leq 0, y \leq 0)$

Note: Testing is black box!

Let's say we make the source code available.

Is it feasible to enumerate test cases to cover all paths?

Assuming two-way branching

Number of branching conditions	Potential number of exectuion paths
1	2 (2^1)
2	4 (2^2)
3	8 (2^3)
..	...
10	1024 (2^{10})
20	1048576 (2^{20})
30	1073741824 (2^{30})

Path complexity is exponential!

A looping program

```
...
while (condition) {
   L
}
...
```

Induces unbounded number of paths

Paths

......
...L...
...LL...
...LLL...
...LLLL...
...LLLLL...
... so forth

Concurrency increases path complexity by multifold. We will examine later.

A foolproof way to prove logic correctness is by use of proofs.

- Think induction, deduction, contrapositive from logic.
- But we need tools to do proofs in automated way.

FORMAL VERIFICATION

(broadly two approaches)

1. Code based

- Suited for proving program is correct.
- Easy to use!

2. Model based

- Suited for proving design is correct.
- Catch errors early!

How does formal verification work?

Source code transformed into logical formulae and inference rules are applied to check if the correctness criteria is met.

Source/Model annotated with specification $\longrightarrow$ **Verification system** $\longrightarrow$

- YES, if able to prove.
- NO, if able to disprove.
- NO RESPONSE

Source	Program source code preferably written modularly.
Model	Blueprint/Design expressed in formal language such as predicate logic or specialized modeling language.
Specification	Embodies correctness criteria in the form of assertions, pre- and post- conditions, loop invariants, etc.
Verification system	Breaks down the proof into smaller steps and applies rules of logic to deduce the validity automatically.

Testing vs. Verification

- Testing uses black-box approach. Verification takes a white-box.
- Testing is a dynamic technique. Verification is usually static.
- Testing tends to be incomplete since each execution covers only one of the many paths. We saw the challenges in covering all paths. In contrast, verification is complete since it uses source code which contains the entire logic.
- Testing is accurate since it is based on real execution. However, verification tends to be approximate in some cases due to abstraction and conservative in conclusion.
- Verification demands exponential effort theoretically. With the rise of computing power and advances in automated theorem proving, it has now become practical to establish proofs by breaking down the verification problem into smaller units. When proof cannot be established in bounded steps, NO RESPONSE is the result.

Setting the expectation

Formal verification tools are work in progress. Although, they have come a long way, they are not adopted by industry fully.

- Testing still rules majority of software development.
- But formal verification can play a complementary role to testing.
- Formal verification tools are common place when it comes to development of critical software.
- Formal verification has achieved great success in hardware domain.

This course will introduce you to two tools.

1. Frama-c: A code-based functional verification tool for C language.
2. SPIN: A model-based behavioral verification tool for models developed using Promela.

Functional verification is concerned with input-output correctness.

- Take an overview of weakest precondition calculus and understand how deductive mechanism is used to prove correctness.
- Introduce ANSI C Specification Language (ACSL) that define basic constructs for correctness specification.
- Learn to work with Frama-c, a practical functional verification tool, that allows correctness criteria to be stated using ACSL and prove correctness of C programs.
- Understand issues in designing concurrent systems and get an overview of model based verification using SPIN/Promela.

The main takeaway from this course for you is to develop deeper insights into subtle issues in programming making you a thoughtful programmer.