Model Checking

Program Correctness

Two approaches to demonstrating that a program does what it's supposed to do:

1. Testing
2. Proof

Testing

- Approach: test "lots of cases"

- (Aside: there is a sub-field of computer science concerned with developing the smallest "complete" set of tests)

- More tests passed, more likely program is defect-free

- But ... can never be *sure* there are no defects — maybe your tests just weren't comprehensive enough

- And: cases that are hard to test likely to be cases hard to program correctly

- *Just when you need testing the most is when it lets you down*

Testing Concurrent Programs

- Testing concurrent programs is extra difficult

- Regarding each thread as a sequence of read/writes to shared variables ... want to test all interleavings

- But since interleaving is controlled by OS scheduler, user cannot arrange arbitrary interleavings

- Consequence: VERY few of possible interleavings are tested

Proving Programs Correct

- Holy Grail of computer science

- Using special **specification language**, describe

1. State of program's variables
2. How each programming language statement uses variables

- Specification language is mixture of mathematics & programming language

Example

- Specification of `for` statement would include (among other things) "when statement is finished, the limit condition is false"

```
int i;
for (i=0; i<100; i++) {
    // body
}
```

- When this `for` is finished, we know that `i` $\geq$ 100

- (Additional reasoning—namely, that `i` increases by exactly 1 each time thru loop—needed to establish more precisely that `i` = 100)

How to Prove a Program Correct

1. Using spec language, characterize how each type of programming language statement—assignment, `for`, etc.—uses program's variables

2. Break program into pieces; e.g., statements or methods

3. Using spec language, write specification for each piece

4. Try to prove that each piece satisfies its specification (i.e., behaves as intended)

Example, I

```
void aMethod() {
// state 1, written in spec language
    Java stmt A;    // have spec for this stmt
// state 2
    Java stmt B;    // have spec for this stmt
// state 3
    Java stmt C;    // have spec for this stmt
// state 4
    return;
}
```

Example, II

- Prove sequence of theorems:

  1. `if (state 1) and (Java statement A) then (state 2)`
  2. `if (state 2) and (Java statement B) then (state 3)`
  3. `if (state 3) and (Java statement C) then (state 4)`

- Once each proven, overall theorem is proven: `aMethod() transforms state 1 to state 4`

Mechanical Proof

- Realistic-length target program generates HUGE number of theorems to prove, each typically small

- Therefore: develop a theorem-proving program & have *it* prove target program correct—the holiest of Holy Grails

- If target program contains a bug, theorem-proving program reports: "the first mini-theorem that cannot be proven is [this one]"

- Fix program then re-run theorem prover

Promise of Program Proof

1. Proof effort is

$$O(number\ of\ lines\ of\ source)$$

whereas complete testing is

$$O(number\ of\ paths\ through\ code)$$

2. A proof is a *proof*, whereas passing a test means only that the test failed to find a bug

- Testing can show absence of bugs only in tested cases, whereas a proof covers ALL cases

Drawbacks of Program Proof, I

- Drawback number 1: precisely specifying program's intended actions is notoriously hard

- Academic assignments not representative of real world—precisely specified to make grading fair/easy & to focus students on recently taught material

- Typical real-word "specification:"

  - Customer request: customers rarely really know what they want; have a few key features in mind & expect software vendor to fill in rest of picture

  - Internal request: "competitor X is kicking our butt with product Y—we have to outdo them!"

Drawbacks of Program Proof, II

- Mathematical proof requires not only precise high-level spec, but also precise spec down to the class/method/statement level

- Doing such a detailed spec & associated proofs usually *much harder than writing the program!*

- Depending on spec language, spec may be as long or longer too

Example, I

Trivial to prove this method, right?

```
int addOne(int x) {

// x = Xinit
    x++;
// x = Xinit+1

    return x;
}
```


Example, II

Wrong!

- If `x` has max value of $2^{31} - 1$, then after `x++` its value is -2^{32}

- Now must write:

```
// x = Xinit
    x++;
// if (Xinit != MAX_VAL)
//    x = Xinit+1
// else
//    x = MIN_VAL
```

- Complicated post-condition becomes pre-condition for next statement—complexity quickly snowballs

Drawbacks of Program Proof, III

- Drawback number 2: Hard to write (tractable) specifications of statements of realistic language

- One reaction: create new programming languages with simple & well defined semantics that map well to mathematics of specification/proof (e.g., side-effect-free functional languages)

- Such languages have proven OK for small programs but not for medium/large ones

- Open question whether one language can satisfy both programmers & theorem provers

Drawbacks of Program Proof, Summary

- Summary: doing the proof often WAY harder than doing the program!

- Put another way: proof effort may be $O(number\ of\ lines\ of\ source)$, but the constant is quite large

- Put yet another way: proof of any well-tested program more likely to have bugs than the program itself

State of the Art of Program Proof

- 30+ years of work in computer science research—field called "formal methods"

- Proof of real programs written in real languages remains a very distant goal

- There have been some partial victories:

  - Theory of programming language semantics
  - Techniques developed for program proof used in smaller & more specific situations, e.g., O-O type checking
  - Functional languages

To Learn More About Formal Methods

CS 643: Formal Verification of Software
taught by Prof. Naumann

Summary of Program Correctness Techniques

Testing is flawed – cannot cover all the cases & the hardest cases to program are also hardest to test

Proof is flawed – WAY too hard

Utility of Formal Methods

Q: Since we presently can't prove non-trivial programs correct, formal methods are a failure, right?

A: Wrong.

- There has been much more progress on restricted problem:

1. Develop a *model* of the program
2. Prove properties of the model
3. Translate model into code

Requirements for a Model

Approach of "prove the model, not the program" is valuable provided that there is simple, accurate correspondence between model and program:

1. Easy to create model corresponding to program's design
2. Easy to prove properties of model
3. Easy to create program corresponding to proven model

Candidate Models

- Many models have been developed

- Each has (dis)advantages for modeling certain types of programs

- Among the oldest & most famous:

  - CCS: Calculus of Communicating Systems (Milner)
  - CSP: Communicating Sequential Processes (Hoare)
  - LOTOS (ISO standard)
  - FSP: Finite State Processes

Spin

We're going to use (a version of) the "Spin" model checking system

Started in 1980 by Gerard Holzmann

Awarded ACM's Software Systems Award in 2001

Spin

Spin poses a learning curve

jSpin is a Java-based GUI front end to Spin

Erigone is a teaching-based simplification of Spin

All use (some subset of) "Promela" language