

# CSC444 - Software Engineering I

Week 8-1

## Correctness proofs

- Construct a formal specification of the program
- Demonstrate that the program satisfies its specification
  - $\{P\}S\{Q\}$  where  $S$  is the program,  $P$  and  $Q$  its pre- and post-conditions respectively
- Apply formal proof to “important” ADTs in your application

## Formal specifications of ADTs

- **Algebraic:** gives semantics implicitly using a set of axioms
  - **type stack:**
    - **functions**
      - create  $\rightarrow$  stack
      - push: stack X int  $\rightarrow$  stack
      - pop: stack  $\rightarrow$  int
    - **axioms**
      - pop(create)=create
      - pop(push(s,i))=s
  - **end stack**

## ... cont'd

- **Model-oriented:** uses abstract object to represent the ADT. The semantics of various functions are given explicitly.
  - **Let** stack= < ...  $x_i$  ... > **where**  $x_i$  is int
  - **invariant**  $0 \leq \text{length}(\text{stack})$
  - **initially** stack= nullseq
  - **function**
    - push (s:stack, x:int)     **pre**  $0 \leq \text{length}(s)$      **post**  $s=s' \sim x$
    - pop (s:stack)             **pre**  $0 < \text{length}(s)$      **post**  $s=\text{leader}(s')$
- Examples: Z, VDM

... cont'd

- Pre and post-conditions
  - need to specify pre and post conditions
  - need to identify an invariant and prove by induction
  - alternative: transformational approach
    - a series of correctness-preserving transformations
    - high-level formal specification is refined into an executable code
    - need wide-spectrum languages

## Coverage-based testing

- The adequacy of testing is expressed in terms of the coverage of the application
- Looking at the control flow graph of a program
  - All-paths
  - All-nodes or statement
  - All-edges or branch
    - extended branch coverage
  - McCabe's or cyclomatic-number

```

1: void insert (a,b,n,x) {
2:   found=false;
3:   for (int i=1; i< n; i++) {
4:     if (a[i]==x)
5:       { found=true; break }
6:   }
7:   if (found)
8:     b[i] ++;
9:   else { n++; a[n]=x; b[n]=1 }
10: }

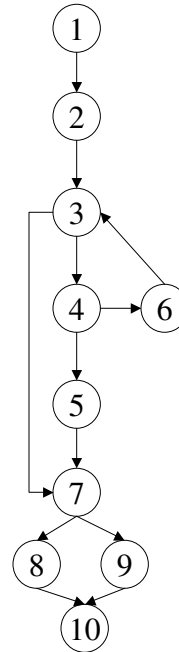
```

$$CV(G) = e - n + p + 1$$

e is the number of edges

n is the number of nodes

p is the number of components



## Fault-based testing

- Aimed at finding a test set with a high ability to detect faults: fault-seeding, mutation testing
- Fault-seeding: to estimate the number of pikes in Lake Soft:
  - catch a number of pikes in Lake Seed, N
  - mark them and throw them into Lake Soft
  - catch a number of pikes, M, in Lake Soft, where M' are from Lake Seed
  - $(M-M') \times N/M'$  estimates the total number of pikes originally in Lake Soft

## ... cont'd

- Artificially add a number of bugs into a program
- Test and determine which ones your test case detected
- If we find many seeded faults and relatively few others, the results can be trusted. The opposite is not true.
- More generally, the more errors you find is an indication of poor quality in that module.

## ... cont'd

- Which test case is better  $T_1$  or  $T_2$ 
  - Transform program P to P'
    - change a - to a +
    - for (int i=0; i<n; i++) becomes for (int i=0; i<n+1; i++)
  - Run test cases  $T_1$  and  $T_2$ 
    - if  $T_1$  produces the same results, it does not discriminate
- Generate many mutants. As soon as a test case generates a different results for one mutant, it is dead
  - Mutation adequacy score=  $D/M$ , where D is the number of dead mutants and M is the total number of mutants

## Underlying story

- Fault seeding: the distribution of pikes from Lake Soft and Lake Seed are the same
- Mutation:
  - Competent programmer hypothesis: write programs that are close to being correct --it differs from a correct version by minor faults
  - Coupling effect hypothesis: tests that can reveal simple faults can also reveal complex faults