Verification

(See related materials in textbook.)
Outline

• What are the goals of verification?
• What are the main approaches to verification?
  – What kind of assurance do we get through testing?
  – How can testing be done systematically?
  – How can we remove defects (debugging)?
• What are the main approaches to software analysis?
  – informal vs. formal
Need for verification

• Designers are fallible even if they are skilled and follow sound principles

• Everything must be verified, every required quality, process and products
  – even verification itself…

• Correctness: must have point of reference
  – Design is correct wrt requirements
  – Code is correct wrt design, requirements
  – Code correctness: does a program work as expected for a given set of inputs
Properties of verification

• May not be binary (e.g., right, wrong)
  – severity of defect is important
  – some defects may be tolerated

• May be subjective or objective
  – e.g., usability

• Even implicit qualities should be verified
  – because requirements are often incomplete
  – e.g., robustness
Approaches to verification

• Experiment with behavior of product
  – sample behaviors via testing
  – goal is to find "counterexamples"
  – dynamic technique

• Analyze product to deduce its adequacy
  – analytic study of properties
  – static technique
Testing and lack of "continuity"

- Testing samples behaviors by examining "test cases"
- Impossible to extrapolate behavior of software from a finite set of test cases
- No continuity of behavior
  - it can exhibit correct behavior in infinitely many cases, but may still be incorrect in some cases
Verification in engineering

- Example of bridge design
- One test assures infinite correct situations
procedure binary-search (key: in element; 
  table: in elementTable; found: out Boolean) is 
begin 
  bottom := table'first; top := table'last; 
  while bottom < top loop 
    if (bottom + top) rem 2 ≠ 0 then 
      middle := (bottom + top - 1) / 2; 
    else 
      middle := (bottom + top) / 2; 
    end if; 
    if key ≤ table (middle) then 
      top := middle; 
    else 
      bottom := middle + 1; 
    end if; 
  end loop; 
  found := key = table (top); 
end binary-search
Goals of testing

• Show the presence of bugs (Dijkstra, 1987)
• If a test does not detect a failure,
  – then CANNOT conclude that software is defect-free
• Still, we need to do testing
  – driven by sound and systematic principles
Goals of testing (cont.)

• Should help isolate errors
  – to facilitate debugging

• Should be repeatable
  – repeating the same experiment, we should get the same results
    • this may not be true because of the effect of execution environment on testing
    • because of *nondeterminism*

• Should be accurate
Theoretical foundations of testing
Definitions (1)

• We view a program to test as a function
  – when invoked with some input $d \in D$
  – produces some output $r \in R$
  – $P: D \rightarrow R$ (may be partial)
  – $P$ (program), $D$ (input domain), $R$ (output domain, i.e., range)

• Correctness defined by an output relation, $O_R$
  – $O_R \subseteq D \times R$
  – $P(d)$ correct if $<d, P(d)> \in O_R$
  – $P$ is correct if all $P(d)$ are correct

• Note: Ghezzi uses OR representation for Output Relation (We use to $O_R$ avoid confusion with logical operator.)
Definitions (2)

• **FAILURE**
  – $P(d)$ is not correct
    • may be undefined (error state) or may be the wrong result

• **ERROR (DEFECT)**
  – anything that may cause a failure
    • typing mistake
    • programmer forgot to test “x = 0”

• **FAULT**
  – incorrect intermediate state entered by program
Definitions (3)

- Test case \( t \)
  - an element of \( D \)
- Test set \( T \)
  - a finite subset of \( D \)
- Test is "successful" (passed) if \( P(t) \) is correct
- Test set "successful" (passed) if \( P \) is correct for all \( t \) in \( T \)

"passed" term used by B. Cheng
Definitions (4)

- **Ideal test set** $T$
  - if $P$ is incorrect, then there is an element of $T$ such that $P(d)$ is incorrect

- *if an ideal test set exists for any program, we could prove program correctness by testing*
Test criterion

- A **test selection criterion C** specifies conditions that must be specified by a test set.
  - C defines finite subsets of domain D (test sets)
    - $C \subseteq 2^D_F$, where $2^D_F$ denotes all finite subsets of D

- A test set T satisfies C if it is an element of C

Example

$C = \{<x_1, x_2, \ldots, x_n> | n \geq 3 \land \exists i, j, k, (x_i < 0 \land x_j = 0 \land x_k > 0)\}$

- $<-5, 0, 22>$ is a test set that satisfies C
- $<-10, 2, 8, 33, 0, -19>$ also does
- $<1, 3, 99>$ does not
Properties of criteria (1)

• **C is consistent**
  – for any pairs $T_1, T_2$ satisfying $C$, $T_1$ is successful iff $T_2$ is successful
  • so either of them provides the “same” information

• **C is complete**
  – if $P$ is incorrect, then there is a test set $T$ of $C$ that is not successful

• **C is complete and consistent**
  – identifies an ideal test set
  – enables correctness to be proved!
Properties of criteria (2)

• **C1 is finer-grained than C2**
  – for any program P
    • for any T1 satisfying C1 there is a subset T2 of T1 which satisfies C2
Properties of definitions

• None is effective, i.e., no algorithms exist to state if a program, test set, or criterion has that property

• In particular, there is no algorithm to derive a test set that would prove program correctness
  – there is no constructive criterion that is consistent and complete
Empirical testing principles

• Attempted compromise between the impossible and the inadequate

• Find strategy to select significant test cases
  – significant=has high potential of uncovering presence of error
Complete-Coverage Principle

- Try to group elements of D into subdomains $D_1, D_2, \ldots, D_n$ where any element of each $D_i$ is likely to have similar behavior
  - $D = D_1 \cup D_2 \cup \ldots \cup D_n$

- Select one test as a representative of the subdomain

- If $D_j \cap D_k = \emptyset$ for all $j, k$ (partition), any element can be chosen from each subdomain

- Otherwise choose representatives to minimize number of tests, yet fulfilling the principle
Complete-Coverage Principle

example of a partition
Testing in the small

• We test individual modules

• **BLACK BOX** (functional) testing
  – partitioning criteria based on the module’s specification
  – tests *what the program is supposed to do*

• **WHITE BOX** (structural) testing
  – partitioning criteria based on module’s internal code
  – tests *what the program does*