Verification

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...

Properties of verification

May not be binary (OK, not OK)
  – 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) mod 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

if we omit this
the routine
works if the else
is never hit!
(i.e. if size of table
is a power of 2)

Goals of testing

Show the presence of bugs (Dijkstra, 1987)
If tests do not detect failures, 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 ∈ D
– produces some output r ∈ R
– P: D → R (may be partial)
– P (program), D (input domain), R (output domain)

Correctness defined by an output relation
– OR ⊆ D × R
– P(d) correct if <d, P(d)> ∈ OR
– P correct if all P(d) are correct
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 if \( P(t) \) is correct
- Test set successful if \( P \) correct for all \( t \) in \( T \)

Definitions (4)

Ideal test set \( T \)
- if \( P \) is incorrect, 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 criterion \( C \) defines finite subsets of \( D \) (test sets)
- \( C \subseteq 2^D \)

A test set \( T \) satisfies \( C \) if it is an element of \( C \)

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

\(<-5, 0, 22> \) 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, there is a test set \( T \) of \( C \) that is not successful

\( C \) is complete and consistent
- identifies an ideal test set
- allows correctness to be proved!

Properties of criteria (2)

\( C_1 \) is finer than \( C_2 \)
- for any program \( P \)
  - for any \( T_1 \) satisfying \( C_1 \) there is a subset \( T_2 \) of \( T_1 \) which satisfies \( C_2 \)
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 D₁, D₂, …, Dᵢ where any element of each Di is likely to have similar behavior.
– D = D₁ ∪ D₂ ∪ … ∪ Dᵢ
Select one test as a representative of the subdomain.
If Dj ∩ Dk = ∅ 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.

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.