

Verification

(Chapter 6)

Outline

- Goals of verification
- Main approaches to verification
 - What kind of assurance do we get through testing?
 - Theoretical foundations of testing
 - How can testing be done systematically?
 - How can we remove defects (debugging)?
- 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
 - process
 - products
 - even verification itself...
- What about formal correctness proofs?

Properties of verification

- May not be binary (OK, not OK)
 - severity of defect is important
 - some defects may be tolerated (see the correctness definition)
- 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

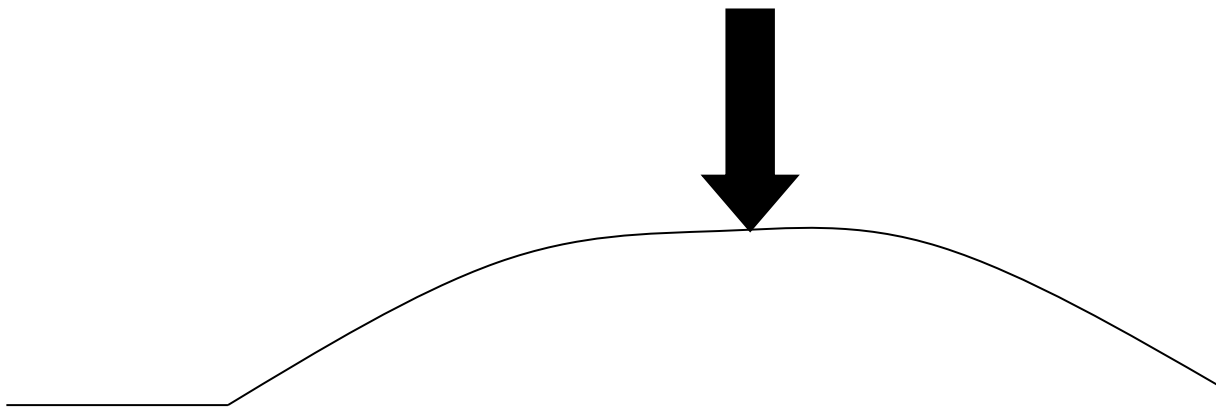
- Experiment with behavior of product
 - sample behaviors via testing
 - goal is to find "counterexamples"
 - dynamic technique
- Analyze product to deduce its adequacy
 - analytic study of properties
 - static technique

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: 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 do  
    if (bottom + top) % 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 while;  
  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

- To show the *presence* of bugs
 - driven by sound and systematic principles
 - If tests *do not* detect failures, we cannot conclude that software is defect-free

```
// test with random x, y
void foo( int x, int y) {
    if x == y then
        print "error";
    else
        ...;
}
```

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
 - “Heisenbugs”
- Should be accurate
 - “System must respond within 1 sec” vs “System must respond within 1 sec regardless of other input events”
 - Formal specs help !

Theoretical foundations of testing

Definitions (1)

- **P** (program), **D** (input domain), **R** (output domain)
 - $P: D \rightarrow R$ (may be partial)
- Correctness defined by **OR** $\subseteq D \times R$ (output requirement)
 - $P(d)$ correct if $\langle d, P(d) \rangle \in \text{OR}$
 - P correct if all $P(d)$ are correct

Examples

- Computing D, R, OR

```
int P(int x) {  
  if x == 2 then  
    return 1;  
  else if x == 4 then  
    return 3;  
  else  
    ERROR;  
}
```

$D = \mathbb{Z}, R = \mathbb{Z}, OR = \{ \langle 2, 1 \rangle, \langle 4, 3 \rangle \}$

```
int P(int x, int y) {  
  if x == y then  
    return 1;  
  else  
    ERROR;  
}
```

$D = \mathbb{Z}, R = \mathbb{Z}, OR = \{ \langle i, i, 1 \rangle \mid i \in \mathbb{Z} \}$

Examples

```
enum E = {A = 1, B = 2};  
E P(E x, E y) {  
    if x > y then  
        return x;  
    else  
        return y;  
}
```

$D = \{1,2\} \times \{1,2\}$

$R = \{1,2\}$

$OR = \{ \langle 1,1,1 \rangle, \langle 1,2,2 \rangle, \langle 2,1,2 \rangle, \langle 2,2,2 \rangle \}$

Examples

```
enum E = {LT = -1, EQ = 0, GT = 1} ;
E P(int x, int y) {
  if x > y then
    return GT;
  else if x == y then
    return EQ;
  else
    return LT;
}
```

$$D = \mathbb{Z} \times \mathbb{Z}$$

$$R = \{-1, 0, 1\}$$

$$\begin{aligned} \text{OR} = \{ \langle i, j, -1 \rangle \mid i \in \mathbb{Z}, j \in \mathbb{Z} \wedge i < j \} \cup \\ \{ \langle i, i, 0 \rangle \mid i \in \mathbb{Z} \} \cup \\ \{ \langle i, j, 1 \rangle \mid i \in \mathbb{Z}, j \in \mathbb{Z} \wedge i > j \} \end{aligned}$$

Definitions (2)

- FAILURE
 - $P(d)$ is not correct
 - for input d , the output is $\neq P(d)$
 - 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

$$C = \{ \langle x_1, x_2, \dots, x_n \rangle \mid n \geq 3 \wedge \exists i, j, k, (x_i < 0 \wedge x_j = 0 \wedge x_k > 0) \}$$

$\langle -5, 0, 22 \rangle$ is a test set that satisfies C

$\langle -10, 2, 8, 33, 0, -19 \rangle$ also does

$\langle 1, 3, 99 \rangle$ does not

Consistent criteria

- C is consistent if for any pairs T1, T2 satisfying C, T1 is successful iff T2 is successful
 - so either of them provides the “same” information

```
int P(int x) {  
    if x == 2 then  
        return 1;  
    else if x == 4 then  
        return 3;  
    else  
        ERROR;  
}
```

Consistent

C = {2,4}

Consistent

C = {6,7}

Inconsistent

C = {1,2}

Complete criteria

- C is complete
 - if P is incorrect, there is a test set T of C that is not successful

```
int P(int x) {  
    if x == 2 then  
        return 1;  
    else if x == 4 then  
        return 3;  
    else  
        ERROR;  
}
```

Complete

C = {1,2}

Incomplete

C = {2,4}

Ideal criteria

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

```
int P(int x) {  
    if x == 2 then  
        return 1;  
    else if x == 4 then  
        return 3;  
    else  
        ERROR;  
}
```

Complete + Consistent
C = {?}

Fine vs coarse test criteria

- C1 is finer than C2
 - for any program P
 - for any T1 satisfying C1 there is a subset T2 of T1 which satisfies C2

```
int P(int x) {  
    if x == 2 then  
        return 1;  
    else if x == 4 then  
        return 3;  
    else  
        ERROR;  
}
```

C1 = {2}

C2 = {2,4}

C1 finer than 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

- Compromise between *impossible* and *inadequate*
- Find strategy to select significant test cases
 - significant=has high potential of uncovering presence of error
- More is not always better !

```
int max(int x, int y) {  
    if x > y then  
        return x;  
    else  
        return x;  
}
```

T1 = {<1,2>}

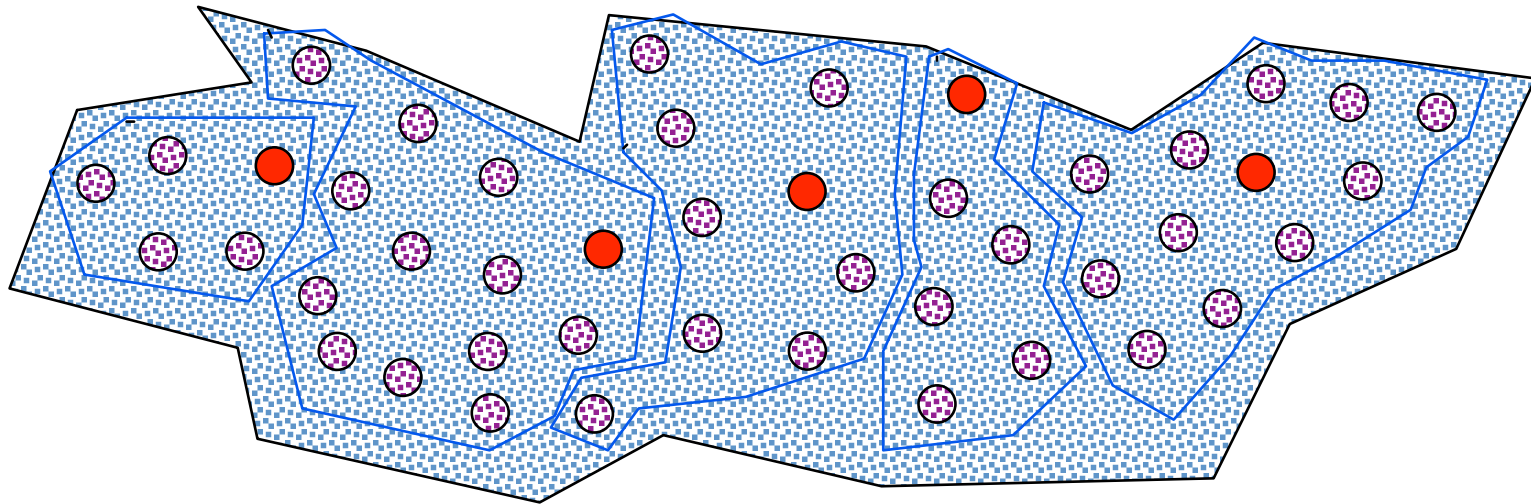
T2 = {<3,2>, <4,3>, <10,5>}

Complete-Coverage Principle

- Try to group elements of D into subdomains D_1, D_2, \dots, D_n where any element of each D_i is likely to have similar behavior
 - $D = D_1 \cup D_2 \cup \dots \cup D_n$
- Select one test as a representative of the subdomain
- If $D_j \cap D_k \neq \emptyset$ 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



Complete-Coverage Principle

example of a partition

```
int P(int x) {  
    if x == 2 then  
        return 1;  
    else if x == 4 then  
        return 3;  
    else  
        ERROR;  
}
```

$D0 = \{i \mid i \in \mathbb{Z}, i < 2\}$

$D1 = \{2\}$

$D2 = \{3\}$

$D3 = \{4\}$

$D5 = \{i \mid i \in \mathbb{Z}, i > 4\}$

Complete-Coverage Principle

example of a minimal partition

```
int P(int x) {  
    if x == 2 then  
        return 1;  
    else if x == 4 then  
        return 3;  
    else  
        ERROR;  
}
```

$D_0 = \{i \mid i \in \mathbb{Z}, i \neq 4 \wedge i \neq 2\}$
 $D_1 = \{2, 4\}$

Reading for next class

Sections 6.3.4.1 and 6.3.4.2