Testing

Testing

- Program testing can be used to show the *presence* of bugs, but never to show their absence. (Dijkstra 1972)
  - random testing is not effective to find flaws (it is "blind", it does not "look for bugs")

Testing and debugging

- Testing should identify the presence of errors
- Debugging must *locate* and *eliminate* errors
- Then a test may be repeated to see if the error has been eliminated
  - repeatability is essential!
  - For concurrent /real-time systems it may be difficult to achieve

Definitions

- P (program), D (input domain), R (output domain)
  - P: D $\rightarrow$ R (may be partial)
- correctness defined by OR $\subseteq$ D $\times$ R
  - P(d) correct if <d, P(d)> $\in$ OR
  - P correct if all P(d) are correct

Definitions

- FAILURE
  - P(d) is not correct
    - may be undefined (error state) or may be the wrong result
- ERROR
  - anything that causes a failure
    - typing mistake
    - programmer forgot to test "x = 0"
- FAULT
  - incorrect intermediate state entered by program

Warning: definitions are not standardized!

Goals

- TESTING
  - Try to increase the probability that the presence of an error results in a failure
- DEBUGGING
  - Try to identify faults as early as possible

Definitions

- 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

Warning: definitions are not standardized!

Definition

- 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 criteria

- A criterion C defines finite subsets of D (test sets)
  - $C \subseteq 2^D$
- A test set T satisfies C if is an element of C
  - Example

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

  <-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 T1, T2 satisfying C

    T1 is successful iff T2 is successful
    - so any of them provides the "same" information
- C is complete
  - if P is incorrect, there is a test set T satisfying C that is not successful
- C is complete and consistent
  - identifies an ideal test set
  - allows correctness to be proved!

Properties of criteria (2)

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

Properties of properties

- None of them 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 criteria

- Attempted compromise between the impossible and the inadequate
- Goal: try to divide D into subdomains D1, D2, …, Dn where any element of each Di is likely to have "similar" behavior
  - $D = D1 \cup D2 \cup \ldots \cup Dn$
- Select one test for each subdomain

complete coverage principle

Choice of test cases

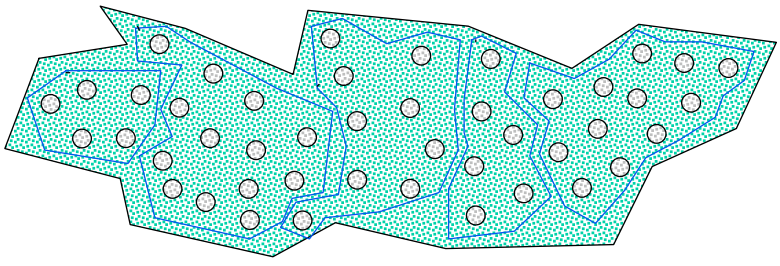
- If $Dj \cap Dk = \varnothing$ for all j, k classes constitute a partition
  - choose any element as a class representative
- If $Dj \cap Dk \neq \varnothing$ then use an element in $D' = Dj \cap Dk$ to reduce tests

Testing in the small

We test individual modules

- BLACK BOX (functional) testing
  - partitioning criteria based on the module's specification
- GLASS (WHITE) BOX (structural) testing
  - partitioning criteria based on module's internal code

Partition



Glass box testing

derives test cases from code
tests what the program does

Structural Coverage Testing

- Simple observation:
  - If significant parts of program structure are not tested, testing is inadequate
- Control flow or control data coverage
  - can be used both to assess adequacy and to generate test cases

Structural Coverage Testing

- Control flow coverage criteria
  - Statement coverage
  - Edge coverage
  - Condition coverage
  - Path coverage
- Data flow coverage (syntactic dependency via def-use)

Coverage-based test evaluation

$$\frac{\text{Number of items covered}}{\text{Total number of items}} \times 100$$

Ideally it should be 1 but can be impossible to achieve

Achieving "all stats" coverage

- Can you find a systematic (algorithmic, mechanical) way to cover all stats?
- If "yes" then there is an algorithm to determine if there is an input that causes termination of any program or otherwise the program does not terminate for any input
  - just consider the "termination statement"
- But this is undecidable!

Edge vs. stat coverage

- Edge coverage is more powerful than statement coverage

```
get(i);
if (i != 0) {
  stats
};
a=b/i;
```

  - 2 test cases vs. 1
  - reveals a fault

Condition vs. edge coverage

- Condition coverage requires that in addition to all edges, all individual components of every conditional be exercised for true and false

```
if ((a != 0) && (b = 1) {stats_1;}
else x = y/a;
```

  - {a=1, b=1}, {a=1, b=0} covers all edges
  - but does not consider the case in which the first conjunct is false, which generates a fault

Path coverage

- All paths cannot be covered
  - in general, they are infinite!
  - one must thus give rules to limit them
    - e.g., a while loop is iterated 0, 1, 2 times
- Even if they were covered, there is no assurance about correctness

Exercise

Specification
The function evaluates GCD of abs. values
The function should raise an exception
if called with zero valued arguments

```
int gcd (int x, y) {
        if (x<0) x= -x;
        if (y<0) y= -y;
        while (x != y) {
                if (x < y)
                        x= x-y;
                else
                        y= y-x;
        }
        return x;
}
```

Good practice:
consider "boundary values" of domains as test candidates

Exercise

```
static int triangle (int a, int b, int c) {
  if ((a==b) && (b==c)) return 1;
  if ((a==b) || (b==c) || (a==c)) return
   2;
  else return 3;
}
```

- Define a test set satisfying the condition coverage criterion

Example

```
1       //read i,j,n
2          while (i<n && j<n) {
3              if (i < j)
4                 i = i + j;
5              else      if (j < i)
6                                 j  = j + i;
7                         else break;
8       }
9       //It continues...
```

(a) Define a test set that satisfies the *edge* criterion but not the *condition* criterion; (R: 3, 3, 10; 3, 4, 10;)

(b) Define a test set that satisfies the *condition* criterion (R: 3, 3, 10; 3, 4, 10; 4, 3, 10)

(c) Define a test suite that covers the following *path*: 1, 2, 3, 4, 8, 2, 3, 5, 6, 8, 2, 9 (R: 3, 4, 10;)

Data-flow based coverage criteria

- Evaluate where a variable is defined

  a = 100;
- where a variable is used

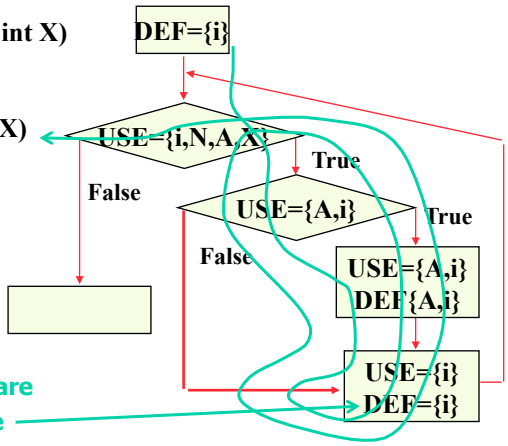
  printf (a);

  a = b + 1; //a is defined; b is used

  i++; //i used and defined

Data Flow Coverage

```
int select(int A[], int N, int X)
{
  int i=0;
  while (i<N and A[i] <X)
  {
    if (A[i]<0)
        A[i] = - A[i];
    i++;
  }
  return(1);
}
```



two iterations of the loop are needed to use the def here

Exercise Def-Use paths: selects paths based on effects on the variables, rather than number of iteration of loops

Def/use coverage

```
0 int select(int A[], int N, int X) {
1   int i=0;
2   while (i<N and A[i] <X) {
3         if (A[i]<0)
4                 A[i] = - A[i];
5         i++;
    }
6   return(1);
}
```

Test set:
<N = 2, X = 5, A[0] = -3, A[1] = -4>

var name	Def/use pairs
i	<1,2><1,3> <1,4><1, 5> <5,2><5,3> <5,4><5, 5>
A	<0,2> <0,3> <0,4> <4,2> <4,3> <4,4>
N	<0,2>
X	<0,2>

The infeasibility problem

- Syntactically indicated behaviors (statements, edges, paths, data flows, etc.) are often impossible
  - unreachable code, edges, paths, etc.
- Adequacy criteria may be impossible to satisfy
  - Coverage approximations
    - example: 95% statement coverage, 80% def -use coverage

Black box testing

derives test cases from specifications

tests what the program is supposed to do

Specs in natural language

- Derivation of test set cannot be automated
- some structure (e.g., organization standards) can help
- guidelines to increase confidence level and reduce discretionality: at least on test case for each
  - subset of "valid" homogeneous data
  - "non valid" (combination of) data
  - boundary data
  - specific data (treated independently, error prone,...)

Exercise

An event queue in a simulation system is a priority queue where events are extracted according to their time -stamps (earliest time-stamps first, last -in among events with the same time -stamp)

- Devise a set of functional test cases for an event queue

A possible solution

Did you consider at least the following cases:

- extracting an event before any insertion
- extracting more events than previously inserted
- extracting the only event in a queue
- many insertions followed by exactly the same number of extractions
- many alternate insertions/extractions
- inserting as many events as the capacity (if defined)
- inserting more events than the capacity (if defined)
- inserting only events with different time-stamps in time-stamp order

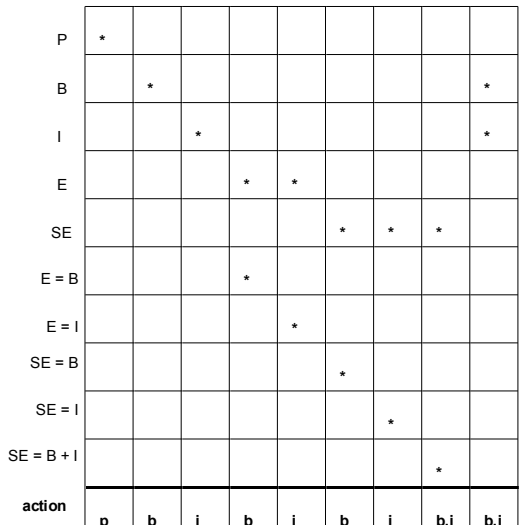
A possible solution (cont.)

- inserting only events with different time-stamps out of time-stamp order
- inserting at least two events with the same time-stamp without interleaving events
- inserting at least two events with the same time-stamp with interleaving events
- insert/extract in/from a non initialized queue (in some languages, e.g., Java, may not be possible)
- ....

Making back-box testing more systematic

Decision table based testing

The word-processor may present portions of text in three different formats: plain text (p), boldface (b), italics (i). The following commands may be applied to each portion of text: make text plain (P), make boldface (B), make italics (I), emphasize (E), superemphasize (SE). Commands are available to dynamically set E to mean either B or I (we denote such commands as E=B and E=I, respectively.) Similarly, SE can be dynamically set to mean either B (command SE=B) or I (command SE=I), or B and I (command SE=B+I.)

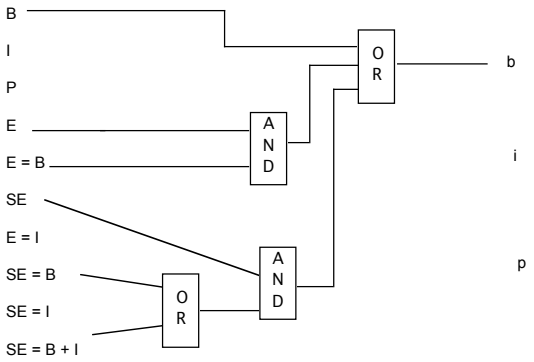


P	*								
B		*							*
I			*						*
E				*	*				
SE						*	*	*	
E = B				*					
E = I					*				
SE = B						*			
SE = I							*		
SE = B + I								*	
action	**p**	**b**	**i**	**b**	**i**	**b**	**i**	**b,i**	**b,i**

Complete coverage criterion

- Exercise at least each column once
- BUT, since columns represent conditions on input data, in worst case 2^n cases for n conditions
  - cause-effect graphs (see next) can help reduce them

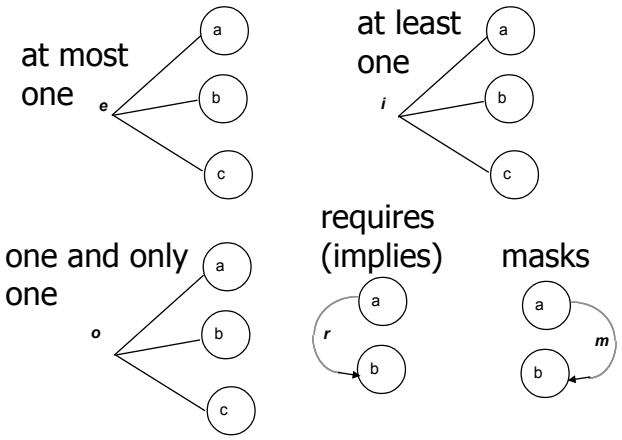
Cause effect graphs

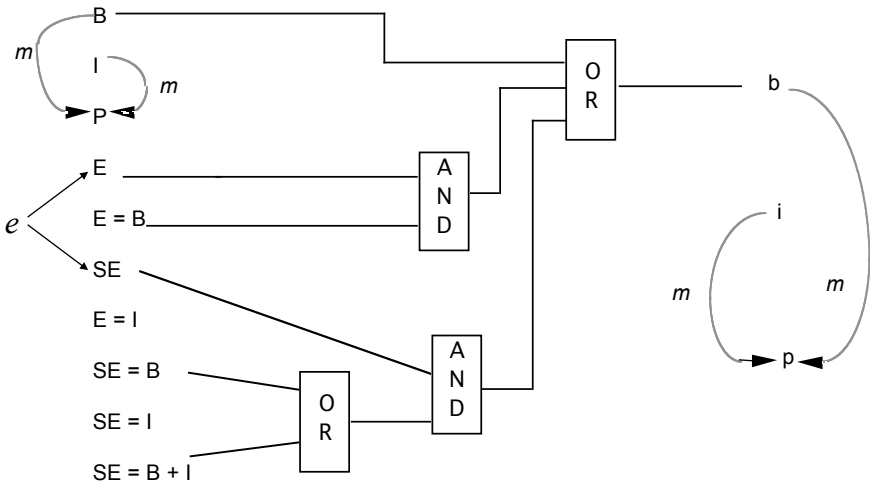


The AND/OR graph represents the correspondence between causes and effects

Further constraints

"Both B and I exclude P (i.e., one cannot ask both for plain text and, say, italics for the same portion of text.) E and SE are mutually exclusive."





X m Y = X implies not Y

Complete coverage criterion

- Generate all possible input combinations and check outputs
- May reduce the number by going backwards from outputs
  - OR node with true output:
    - use input combinations with only one true input
  - AND node with false output:
    - use input combinations with only one false input

Syntax driven testing

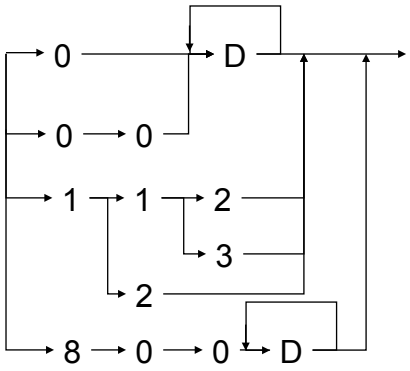
- Given the syntactic structure of input, generate tests that cover all rules

```
<expression> ::= <expression> + <term> |
                 <expression> - <term> | <term>
<term> ::= <term> * <factor>
         | <term> / <factor> | <factor>
<factor> ::= ident | ( <expression>)
```

- The specification is formal, and test generation can be automated

Syntax-driven testing

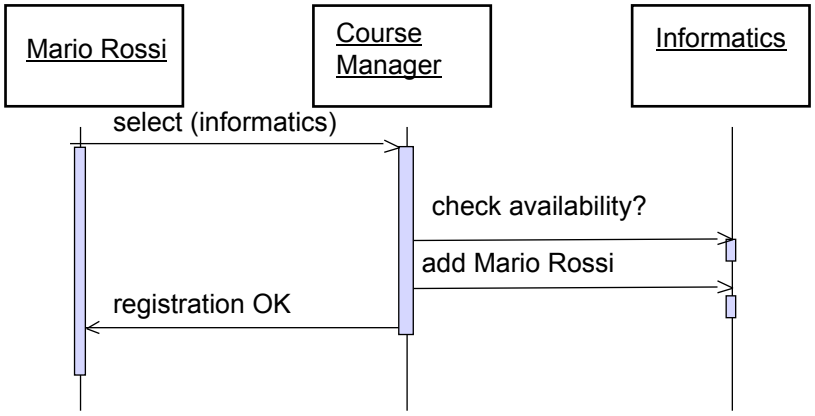
<Tel>::((0|00)(D)$^{5..9}$)|(1(2|1(2|3)))+(800(D$^{5..9}$))



Test cases:
Normal cases:
055675
0233467899
05654665
0013436
0069870780
00767569765
112
113
12
80049321
800567547
800463263264

Test cases:
Special cases:
9865776657
7876
09876543245
0987
009897776548
00987
1127
1139
126
8009877
8008888888888

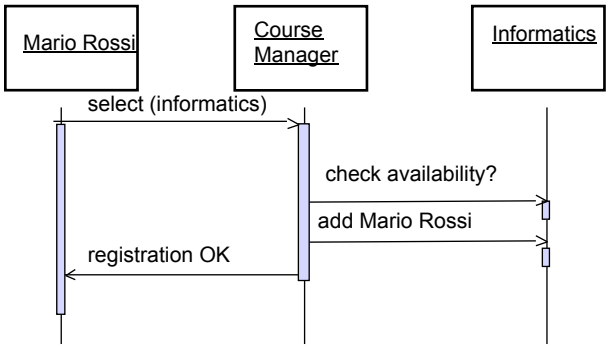
Deriving test cases from sequence diagrams



Approach

- Sequence diagrams define possible sequences of messages
- They should show the frequent and particular cases
- Test cases generation approach
  - First step: Generate a test case for each interaction diagram
  - Next steps: For each diagram identify the possible alternatives and for any alternative select other cases

Example



Alternatives:
- Mario Rossi
  - Incorrect name
  - Student already in the attendee list
  - Student not allowed to select a new course
  - …..
- select informatics
  - Informatics does not exist
  - Informatics is not available
  - …..

White vs black box

Black box

- tests what the program should do
- it depends on the specification notation
- it scales up (different techniques at different granularity levels)
- it cannot distinguish between different implementations for same specs

White box

- tests what the program does
- it is based on control or data flow coverage
- it does not scale up (mostly applicable at unit and integration testing level)
- it cannot reveal errors due to missing parts

The budget coverage criterion

- Industry's answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached

The oracle problem

How to inspect the results of executing test and reveal failures

- Oracles are required at each stage of testing
- Automated test oracles are required for running large amounts of tests
- Oracles are difficult to design - no universal recipe

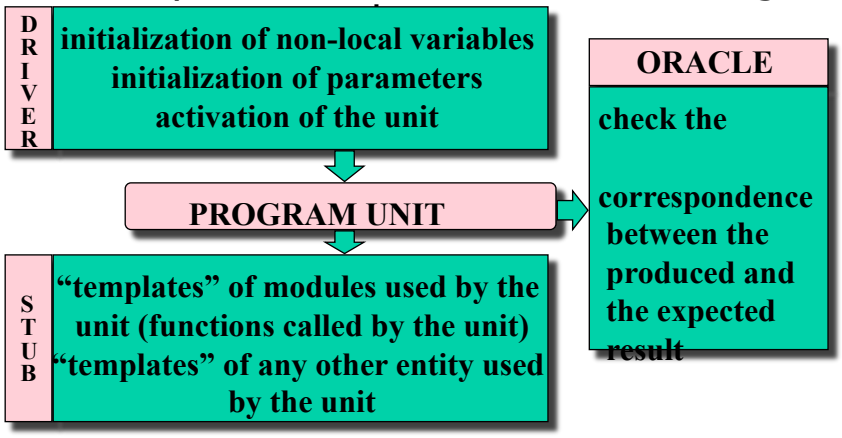
Module testing in context

- Must create the context for executing the unit
- DRIVER
  - must initialize parameters and non local variables
  - must call the unit
- STUB
  - must provide some replacements for the called units

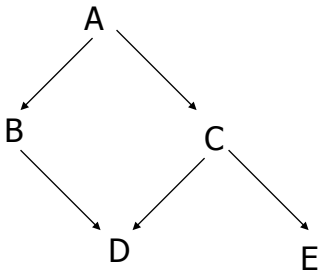
Create Scaffolding

Goal

To setup the environment for executing



Integration testing and USES relation



If integration and test proceeds bottom up only need drivers

Regression Testing

Testing a new version (release): how can we minimize effort using results of testing of previous versions?

- On a previous release:
  - save scaffoldings (drivers,stubs,oracles)
  - record test cases (<inputs,outputs>)
- On the new release:
  - keep track of changes
  - evaluate impact of changes

Test Documentation

- Must be an organization standard
- It must include at least
  - **test set documentation**
    - software tested
    - version
    - goal
    - overall results
    - author
  - **test case documentation**
    - goal
    - "environment" (driver, stub,oracle)
    - input
    - expected output
    - actual output
    - result
    - observations

Tool Support

- Planning and Management
  - activity planning
  - error tracking
  - repository management
  - report generation
- Test execution
  - (code analysis)
  - (partial) generation of scaffoldings
  - capture and playback
  - record of results of test runs

JUnit

Tool Support (cont….)

- Coverage Analysis
  - simple structural coverage (statement |branch|path)
  - graphical (incremental) visualization
  - report generation
- Regression testing
  - re-execution of test suites
  - comparison with previous runs
  - report generation

Introduction to JUnit

see also JUnit Cookbook

http://junit.sourceforge.net/doc/cookbook/cookbook.htm

An example

```
class Money {
        private int fAmount;
        private String fCurrency;
        public Money(int amount, String currency) {
                fAmount= amount;
                fCurrency= currency;
        }
        public int amount() {
                return fAmount;
        }
        public String currency() {
                return fCurrency;
        }
}
```


then you add a new method

When you add two Money of the same currency, the resulting Money has as its amount the sum of the other two amounts.

```
public Money add(Money m) {
        return new Money(amount()+m.amount(),
                                        currency());
}
```

A simple test case

```
@Test public void simpleAdd() {
        Money m12CHF= new Money(12, "CHF");
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);
        assertTrue(expected.equals(result));
}
```

NOTE: here test case = test set

General structure

```
import org.junit.*;
import static org.junit.Assert.*;
public class MyTests {
        @Test
        ….
        @Test
        ….
        ….
}
```

@Test is a Java annotation which indicates a test method

References

- C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 2003.
- M. Pezzè, M. Young *Software Testing and Analysis: Process, Principles, and Techniques*. J. Wiley and Sons, 2007.