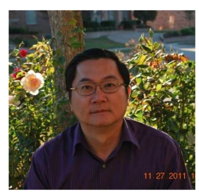
STAR Laboratory of Advanced Research on Software Technology

Requirements-based Test Generation for Functional Testing

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch



- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference (*IEEE International Conference on Software Security and Reliability*) (http://paris.utdallas.edu/sere13)

Two Techniques for Test Generation

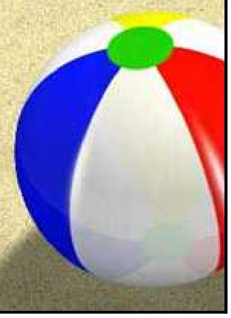
- Equivalence Class partitioning
- Boundary value analysis

} Essential *black-box* techniques for generating tests for *functional testing*

Functional Testing

- Testing a program/sub-program to determine whether it functions as planned
- A *black-box based* testing against the operational (i.e., functional) requirements.
- Testing the *advertised features* for correct operation
- Verifying a program for its *conformance to all functional specifications*
- Entailing the following tasks
  - *Test generation*
  - *Test execution*
  - *Test assessment*

Equivalence Class Partitioning

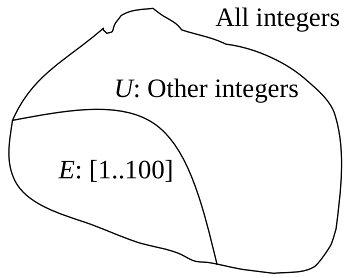


Example I (1)

- Consider an application that takes an integer as input
- Let us suppose that the only legal values are in the range [1..100]
- Which input value(s) will you use to test this application?

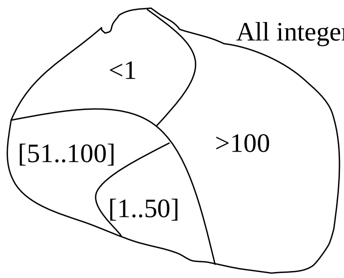
Example I (2)

- The set of input values can be divided into
  - A set of *expected, or legal, inputs* (*E*) containing all integers in the range [1..100]
  - A set of *unexpected, or illegal, inputs* (*U*) containing the remaining integers



Example I (3)

- Assume that the application is required to process all values in the range [1..50] in accordance with requirement R_1 and those in the range [51..100] according to requirement R_2.
  - *E* is divided into two regions depending on the expected behavior.
- Also assume that all invalid inputs less than 1 are to be treated in one way while all greater than 100 are to be treated differently.
  - This leads to a subdivision of *U* into two categories.

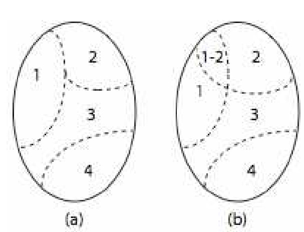


Example I (4)

- *How many input values should we use for testing the application* **?**

Equivalence Partitioning

- Test selection using *equivalence partitioning* allows a tester to divide the input domain into *a relatively small number of sub-domains*.
- The sub-domains are *disjoint.*
- Each subset is known as an *equivalence class.*
- The four subsets shown in (a) constitute a partition of the input domain *while the subsets in (b) are not*.



Quiz

- *What if there is more than one input variable?*

Unidimensional Partitioning

- One way to partition the input domain is to consider *one input variable at a time*. Thus each input variable leads to a partition of the input domain.
- We refer to this style of partitioning as *unidimensional* equivalence partitioning or simply *unidimensional partitioning*.
- This type of partitioning is commonly used.

Multidimensional Partitioning

- Another way is to consider the input domain I as the *set product of the input variables* and define a relation on I. This procedure creates *one partition consisting of several equivalence classes*.
- We refer to this method as *multidimensional* equivalence partitioning or simply *multidimensional partitioning*.

Example **II (1)**

- Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$.
- How many pairs of (x, y) should we use to test this application?

Example II (2)

- Using Unidimensional Partitioning

E1: $x<3$ E2: $3 \le x \le 7$ E3: $x>7$ ← *y* ignored.

E4: $y<5$ E5: $5 \le y \le 9$ E6: $y>9$ ← *x* ignored.

Example II (3)

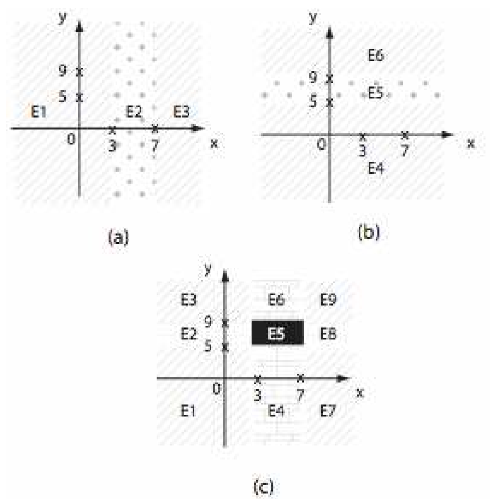
- Using Multidimensional Partitioning

E1: $x<3$, $y<5$ E2: $x<3$, $5 \le y \le 9$ E3: $x<3$, $y>9$

E4: $3 \le x \le 7$, $y<5$ E5: $3 \le x \le 7$, $5 \le y \le 9$ E6: $3 \le x \le 7$, $y>9$

E7: $x>7$, $y<5$ E8: $x>7$, $5 \le y \le 9$ E9: $x>7$, $y>9$

Example II (4)



Equivalence Classes based on Program Output (1)

- In some cases the equivalence classes are *based on the output generated by the program*.
- For example, suppose that a program outputs an integer.
- It is worth asking: “Does the program ever generate a 0? What are the maximum and minimum possible values of the output?”
- These two questions lead to two the following equivalence classes based on outputs:

Equivalence Classes based on Program Output (2)

- E1: Output value v is 0
- E2: Output value v is the maximum possible
- E3: Output value v is the minimum possible
- E4: All other output values

- Based on the *output equivalence* classes one may now derive equivalence classes *for the inputs.* Thus each of the four classes given above might lead to one equivalence class consisting of inputs.

More examples . . . .

Equivalence Classes for variables : Range

Equivalence Classes	Example	
	Constraints	Classes
One class with values *inside* the range and two with values *outside* the range.	speed ∈ [60..90]	{50}, {75}, {92}
	area: float area≥0.0	{{-1.0}, {15.52}}
	age: int	{{-1}, {56}, {0}}

Equivalence Classes for variables : String

Equivalence Classes	Example	
	Constraints	Classes
At least one containing all *legal* strings and one all *illegal* strings based on any constraints.	firstname: string	{{ε}, {Sue}, {Loooong Name}}

Equivalence Classes for variables : Enumeration

Equivalence Classes	Example	
	Constraints	Classes
Each value in a *separate* class	autocolor:{red, blue, green}	{{red,} {blue}, {green}}
	X:boolean	{{true}, {false}}

Equivalence Classes for variables **: *Array***

Equivalence Classes	Example	
	Constraints	Classes
One class containing all *legal* arrays, one containing the *empty* array, and one containing a *larger than* expected array.	int [ ] aName = new int[3];	{[ ]}, {[-10, 20]}, {[-9, 0, 12, 15]}

Equivalence Classes for Compound Data Type **(1)**

- Arrays in Java and structures in C++/C, are *compound types*. Such input types may arise while testing components of an application such as a function or an object.
- While generating equivalence classes for such inputs, one must *consider legal and illegal values for each component of the structure*.
- The next two examples illustrate the derivation of equivalence classes for an input variable that has a compound type.

Equivalence Classes for Compound Data Type (2)

- **struct** transcript

```
{
    string fName;          // First name
    string lName;          //  Last name
    string studentID       // 9 digits
    string cTitle [200];   // Course titles
    char grades [200];     // Letter grades corresponding to course titles
}
```

- *Derive equivalence classes for each component of R and combine them!*

Equivalence Classes for Compound Data Type (3)

- Consider a procedure *P* in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee record consists of the following items with their respective types and constraints:

ID: `int`;	`ID` is 3-digits long from 001 to 999.
name: `string`;	`name` is 20 characters long; each character belongs to the set of 26 letters and a space character.
rate: `float`;	`rate` varies from $5 to $10 per hour; rates are in multiples of a quarter.
hoursWorked: `int`;	`hoursWorked` varies from 0 to 60.

- *Calculate the size of the input domain*

Systematic Procedure for Equivalence Partitioning

- 1. *Identify the input domain*: Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.
- 2. *Equivalence classing*: Partition the set of values of each variable into *disjoint subsets*
- 3. *Combine equivalence classes*: This step is usually omitted and the equivalence classes defined for each variable are *directly used* to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.
- 4. *Identify infeasible equivalence classes*: An infeasible equivalence class is one that contains a combination of *input data that cannot be generated during test*. Such an equivalence class might arise due to several reasons.

Example III (1)

- Consider that wordcount method takes a word *w* and a filename *f* as input and returns the number of occurrences of *w* in the text contained in the file named *f*. An exception is raised if there is no file with name *f*. Using the partitioning method described in the previous example, we obtain the following equivalence classes.

Example III (2)

Equivalence class	*w*	*f*
E_1	non-null	exists, not empty
E_2	non-null	does not exist
E_3	non-null	exists, empty
E_4	null	exists, not empty
E_5	null	does not exist
E_6	null	exists, empty

Example III (3)

- The number of equivalence classes without any knowledge of the program code is 2, whereas the number of equivalence classes on the previous slide is 6.
- An experienced tester will likely derive the six equivalence classes given above, and perhaps more, even before the code is available

Quiz

- How many equivalence classes do we need for the *wordcount* program?

GUI Design and Equivalence Classes (1)

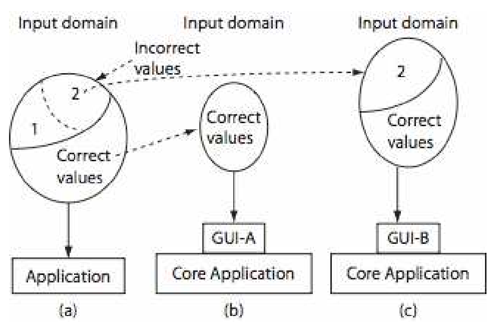
- While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must *account for the possibility of errors in data entry.*

- Example: An application places a constraint on an input variable *x* such that it can assume integral values in the range 3..7. However, testing must *account for the possibility that a user may inadvertently enter a value for x that is out of range.*

GUI Design and Equivalence Classes (2)

- Suppose that all data entry to the application is via a GUI front end. Suppose also that the *GUI offers exactly five correct choices to the user for x.*
- In such a situation it is impossible to test the application with a value of *x* that is out of range. Hence only the correct values of *x* will be input. See figures on the next slide.

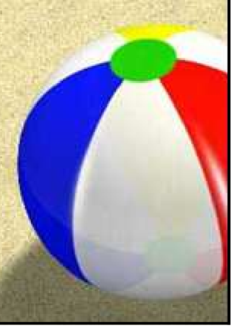
GUI Design and Equivalence Classes (3)



Program Behavior and Equivalence Classes

- The equivalence classes are created assuming that the program *behaves the same on all elements* (i.e., tests) *within a class*.
- This assumption allows the tester to *select exactly one test case from each equivalence class* to test the program.
- Is this assumption correct?
- If yes, why?
- If no, how to improve the test set?

Boundary Value Analysis



Errors at the Boundaries

- Experience indicates that programmers make mistakes in processing values *at and near the boundaries of equivalence classes.*
- For example, suppose that method *M* is required to compute a function f_1 when $x \leq 0$ is true and function f_2 otherwise. Also assume that $f_1(0) \neq f_2(0)$
- However, *M* has an error due to which it computes f_1 for $x < 0$ and f_2 otherwise.
- Obviously, this fault can be revealed when *M* is tested against $x = 0$, but not if the input test set is, for example, {-4, 7} derived using equivalence partitioning.
- In this example, the value x=0, lies at the boundary of the equivalence classes $x \leq 0$ and $x > 0$.

Equivalence Partitioning & Boundary Value Analysis

- While equivalence partitioning selects tests from *within equivalence classes,* boundary value analysis focuses on tests *at and near the boundaries of equivalence classes.*
  - Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.
- Certainly, tests derived using either of the two techniques may overlap.

Boundary Value Analysis : Procedures

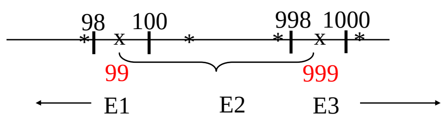
- Partition the input domain using unidimensional partitioning. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.
- Identify the boundaries for each partition. Boundaries may also be identified using special relationships among the inputs.
- Select test data such that each boundary value occurs in at least one test input.

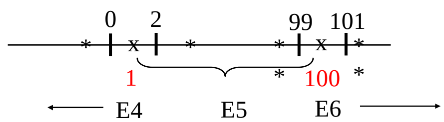
BVA Example : Step 1 – Create Equivalence Classes

- Assuming that a program takes two variables as input: code must be in the range 99..999 and quantity in the range 1..100
  - Equivalence classes for code
    - E1: values less than 99
    - E2: values in the range
    - E3: values greater than 999
  - Equivalence classes for quantity
    - E4: values less than 1
    - E5: values in the range
    - E6: values greater than 100

BVA Example : *Step* 2 – *Identify Boundaries*

- Boundaries are indicated with an x.





BVA Example : *Step* 3 – *Construct Test Set*

- Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary.

T={ t_1: (code=98, quantity=0),
t_2: (code=99, quantity=1),
t_3: (code=100, quantity=2),
t_4: (code=998, quantity=99),
t_5: (code=999, quantity=100),
t_6: (code=1000, quantity=101)
}

Illegal values of *code* and *quantity* included

- Quiz: unidimensional partitioning versus multidimensional partitioning

Equivalence Class Partitioning
versus
Statement Coverage

Example: Identify the Type of a Triangle (1)

- A program *P* takes an input of three integers *a*, *b* and *c*, and returns the type of the triangle corresponding to three sides of length *a*, *b*, and *c*, respectively.
- Quiz:
  - How to generate a test set based on *Equivalence Class Partitioning* to achieve the highest statement coverage possible?

Example: Identify the Type of a Triangle (2)

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right        :  area = b*c / 2;
    equilateral  :  area = a*a * sqrt(3)/4;
    otherwise    :  s = (a+b+c)/2;
                    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

Question:

What is the statement coverage of your test set?

Boundary Value Analysis
Versus
Decision Coverage

Complement between BVA and Decision Coverage

- Test cases generated based on Boundary Value Analysis improve decision coverage.
- Similarly, test cases that achieve high decision coverage also cover some boundary values.
- Examples
  - If ($x \leq 0$) {…..}
    - BVA: {$x_1 = 0$; $x_2 = 1$; $x_3 = -1$}
    - Together, x_1, x_2 and x_3 give 100% decision coverage.
  - If ($y == 3$) {…..}
    - {$y_1 = 3$ and y_2 = a value different from 3} gives 100% decision coverage.
    - At least one of the boundary value (y = 3) is covered.