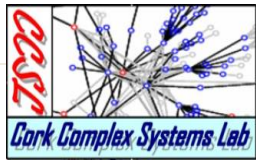
Large-Scale Application Development and Integration

Requirements-Based Testing

Prof. Gregory Provan

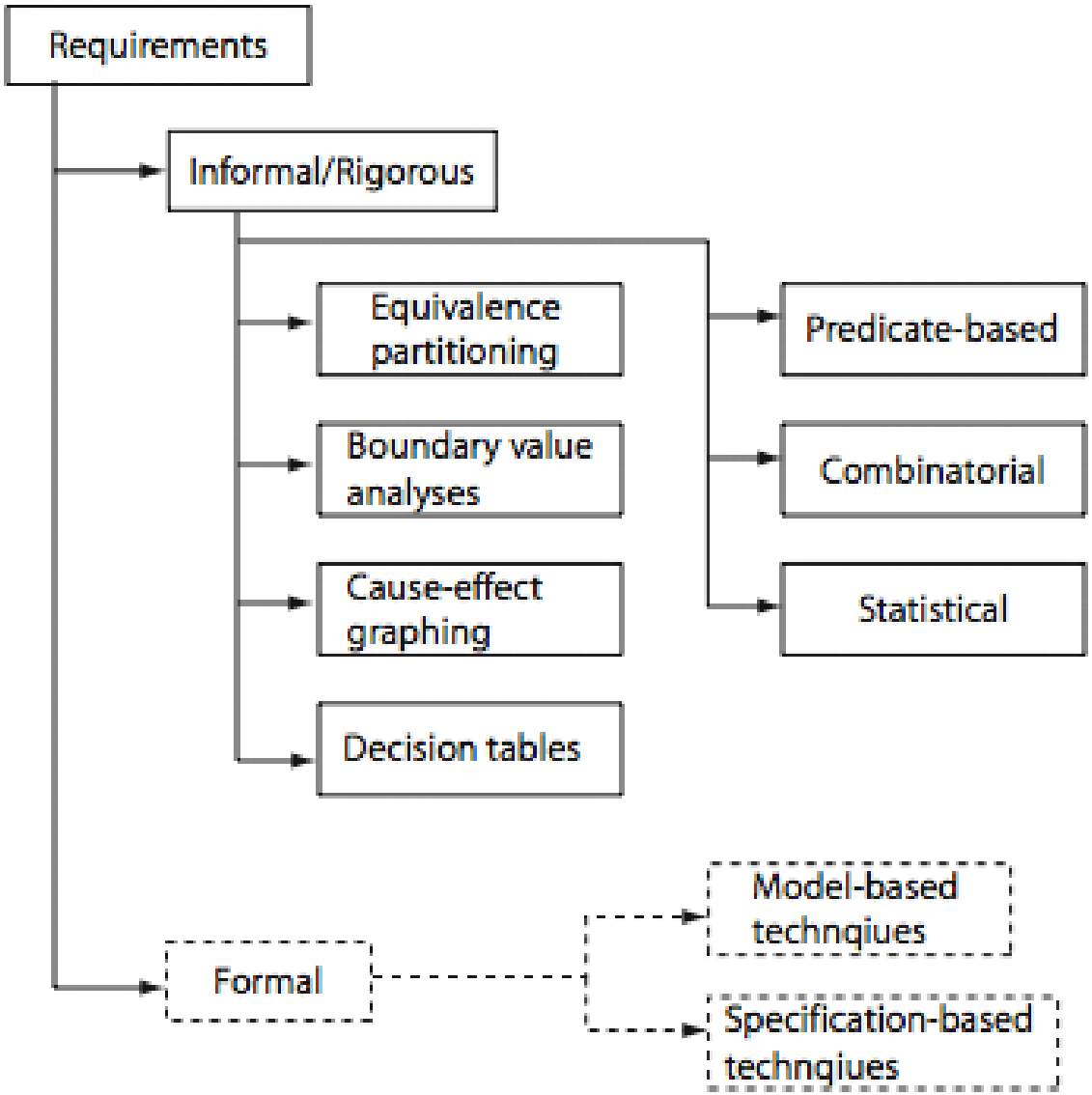


Learning Objectives

- Equivalence class partitioning
- Boundary value analysis
- Test generation from predicates

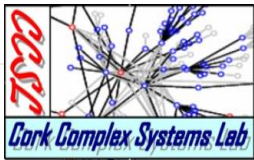
Essential black-box techniques for generating tests for functional testing.

Test generation techniques



Requirements for Software Design

- Each of these testing approaches can also be used for design
- Consider “testing” as part of the design process
- Example: equivalence partitioning
  - Divides parameter-space into “testable” subsets
  - In design, you must specify the “parameter regions” where your code is applicable
  - Example: sensor data analysis



Applications of test generation techniques

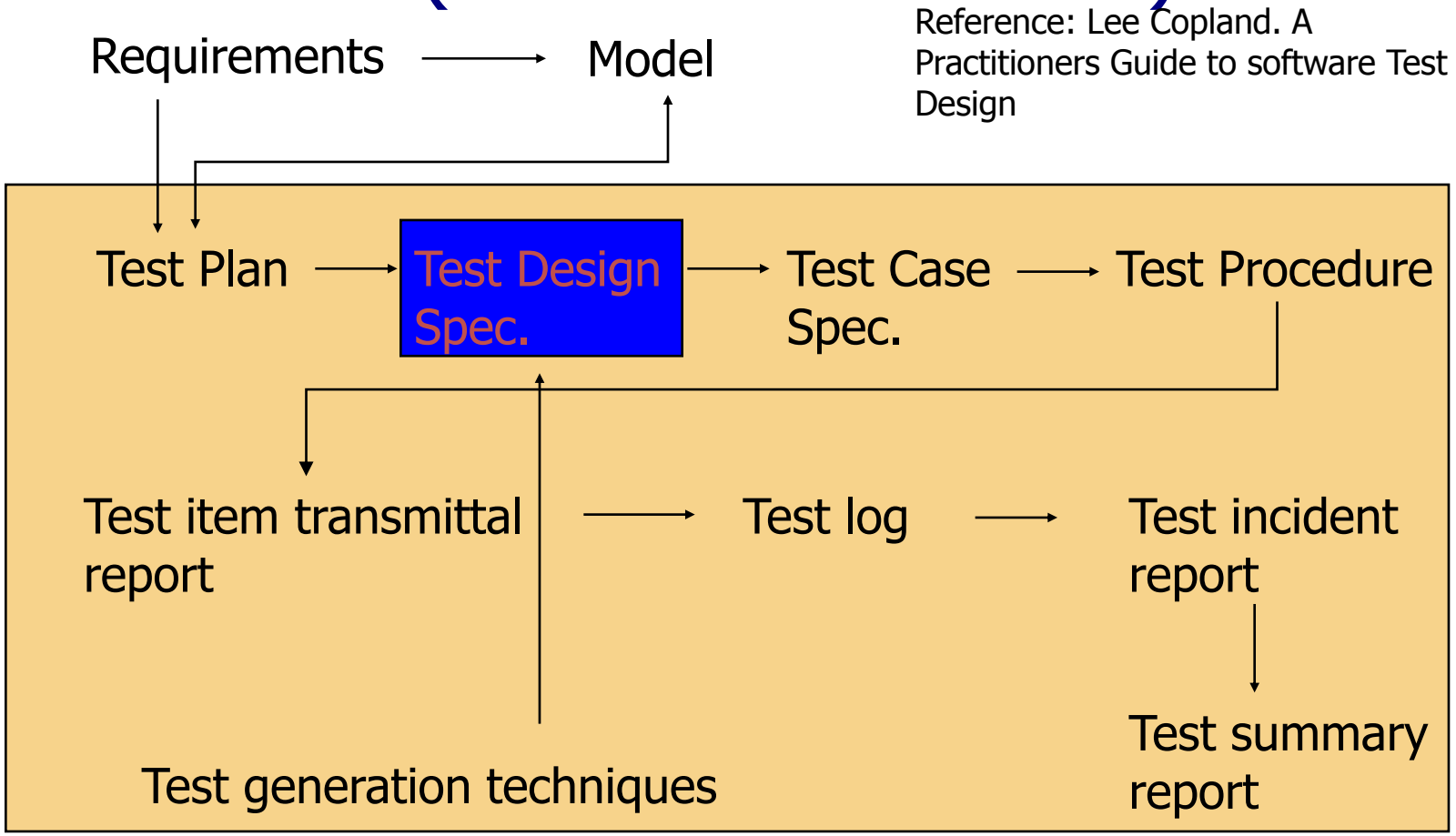
Black-box testing

- no knowledge of the code itself

Techniques are useful during functional testing

- objective is to test whether or not an application, unit, system, or subsystem, correctly implements the functionality as per the given requirements

Functional Testing: Test Documents (IEEE829 Standard)



Reference: Lee Copland. A Practitioners Guide to software Test Design

Functional Testing: Documents

Test Plan: Describe scope, approach, resources, test schedule, items to be tested, deliverables, responsibilities, approvals needed.

Could be used at the system test level or at lower levels.

Test design spec: Identifies a subset of features to be tested and identifies the test cases to test the features in this subset.

Test case spec: Lists inputs, expected outputs, features to be tested by this test case, and any other special requirements e.g. setting of environment variables and test procedures. Dependencies with other test cases are specified here. Each test case has a unique ID for reference in other documents.

Functional Testing: Documents (contd)

Test procedure spec: Describe the procedure for executing a test case.

Test transmittal report: Identifies the test items being provided for testing, e.g. a database.

Test log: A log observations during the execution of a test.

Test incident report: Document any special event that is recommended for further investigation.

Test summary: Summarize the results of testing activities and provide an evaluation.

Black-Box Test Generation

Techniques considered

- equivalence partitioning,
- boundary value analysis,
- cause effect graphing, and
- predicate based test generation.

Each of these test generation techniques is a black-box technique and useful for generating test cases during functional testing.

The test selection problem

Requirements and test generation

Requirements serve as the starting point for the generation of tests. During the initial phases of development, requirements may exist only in the minds of one or more people.

These requirements, more aptly ideas, are then specified rigorously using modeling elements such as use cases, sequence diagrams, and statecharts in UML.

Rigorously specified requirements are often transformed into formal requirements using requirements specification languages such as Z, S, and RSML.

Test selection problem

Let D denote the input domain of a program P. The test selection problem is to select a subset T of tests such that execution of P against each element of T will reveal all errors in P.

In general there does not exist any algorithm to construct such a test set. However, there are heuristics and model based methods that can be used to generate tests that will reveal certain type of faults.

Test selection problem (contd.)

The challenge is to construct a test set $T \subseteq D$ that will reveal as many errors in P as possible. The problem of test selection is difficult due primarily to the size and complexity of the input domain of P.

Exhaustive testing

The large size of the input domain prevents a tester from exhaustively testing the program under test against all possible inputs. By ``exhaustive'' testing we mean testing the given program against every element in its input domain.

The complexity makes it harder to select individual tests.

Large input domain

Consider program P that is required to sort a sequence of integers into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of pr consists of all possible sequences of integers in the range [-32768, 32767].

If there is no limit on the size of the sequence that can be input, then the input domain of P is infinitely large and P can never be tested exhaustively. If the size of the input sequence is limited to, say $N_{max}>1$, then the size of the input domain depends on the value of N.

Calculate the size of the input domain.

Complex input domain

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.

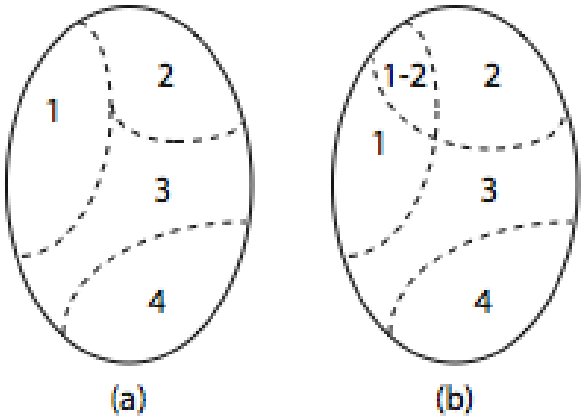
Equivalence class partitioning

Equivalence partitioning

Test selection using equivalence partitioning allows a tester to subdivide the input domain into a relatively small number of sub-domains, say N>1, as shown (next slide (a)).

In strict mathematical terms, the sub-domains by definition are disjoint. The four subsets shown in (a) constitute a partition of the input domain while the subsets in (b) are not. Each subset is known as an equivalence class.

Subdomains



Program behavior and equivalence classes

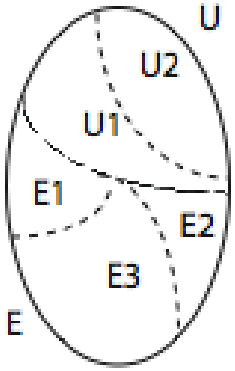
The equivalence classes are created assuming that the program under test exhibits the same behavior on all elements, i.e. tests, within a class.

This assumption allow the tester to select exactly one test from each equivalence class resulting in a test suite of exactly N tests.

Faults targeted

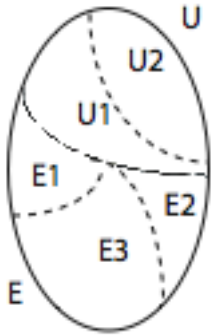
The entire set of inputs to any application can be divided into at least two subsets: one containing all the expected, or legal, inputs (E) and the other containing all unexpected, or illegal, inputs (U).

Each of the two subsets, can be further subdivided into subsets on which the application is required to behave differently (e.g. E1, E2, E3, and U1, U2).



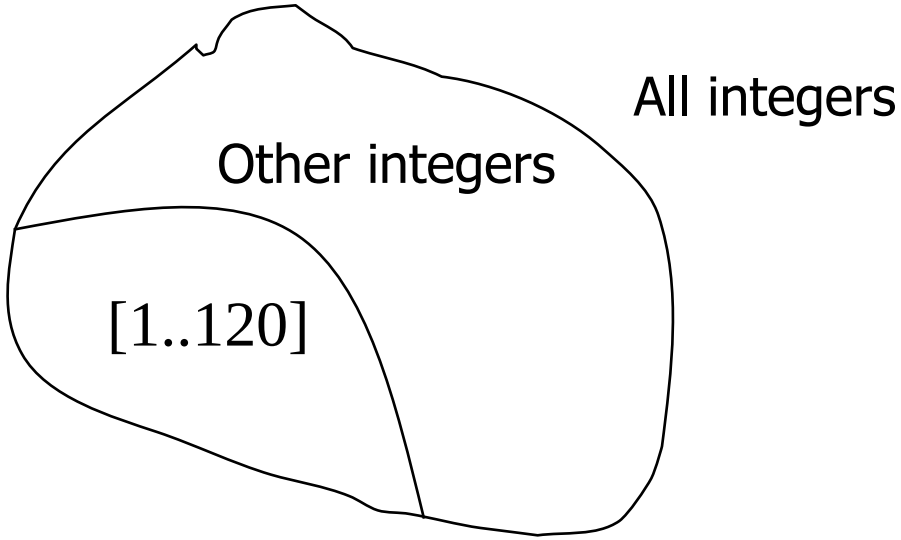
Faults targeted (contd.)

Equivalence class partitioning selects tests that target any faults in the application that cause it to behave incorrectly when the input is in either of the two classes or their subsets.



Example 1

Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.

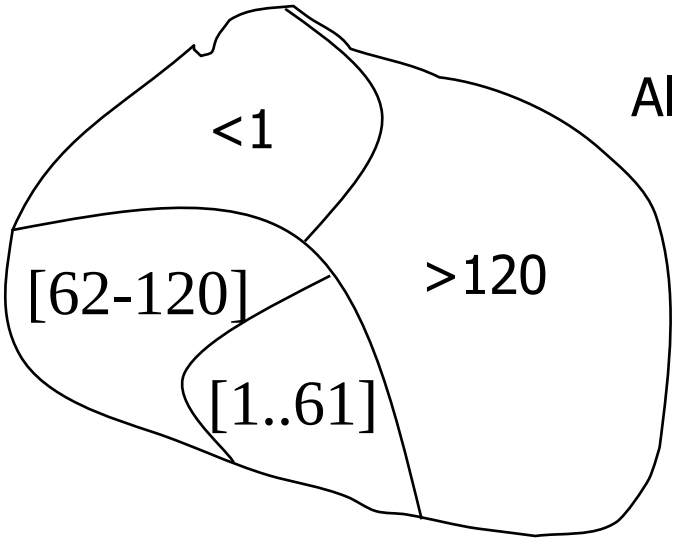


Example 1 (contd.)

Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2.
Thus E is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.

Example 1 (contd.)



All integers

Example 1 (contd.)

Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e. two regions containing expected inputs and two regions containing the unexpected inputs.

It is expected that any single test selected from the range [1..61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62..120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

Effectiveness

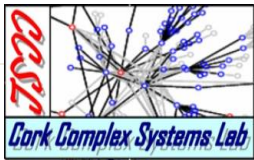
The effectiveness of tests generated using equivalence partitioning for testing application A, is judged by the ratio of the number of faults these tests are able to expose to the total faults lurking in A.

As is the case with any test selection technique in software testing, the effectiveness of tests selected using equivalence partitioning is less than 1 for most practical applications. The effectiveness can be improved through an unambiguous and complete specification of the requirements and carefully selected tests using the equivalence partitioning technique described in the following sections.

Example 2

This example shows a few ways to define equivalence classes based on the knowledge of requirements and the program text.

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.



Example 2 (contd.)

```
begin
    String w, f
    Input w, f
    if (not exists(f) {raise exception; return(0);}
    if(length(w)==0)return(0);
    if(empty(f))return(0);
    return(getCount(w,f));
end
```

Using the partitioning method described in the examples above, we obtain the following equivalence classes.

Example 2 (contd.)

Equivalence class	w	f
E1	non-null	exists, not empty
E2	non-null	does not exist
E3	non-null	exists, empty
E4	null	exists, not empty
E5	null	does not exist
E6	null	exists, empty

Example 2 (contd.)

Note that the number of equivalence classes without any knowledge of the program code is 2, whereas the number of equivalence classes derived with the knowledge of partial code is 6.

Of course, an experienced tester will likely derive the six equivalence classes given above, and perhaps more, even before the code is available

Equivalence classes based on program output

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 (contd.)

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.

Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	Classes
One class with values inside the range and two with values outside the range.	speed $\in$[60..90]	{50}, {75}, {92}
	area: float area≥0.0	{{-1.0}, {15.52}}
	age: int	{{-1}, {56}, {132}}
	letter:bool	{{J}, {3}}

Equivalence classes for variables: strings

Eq. 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

Eq. Classes	Example	
	Constraints	Classes
Each value in a separate class	autocolor:{red, blue, green}	{{red,} {blue}, {green}}
	up:boolean	{{true}, {false}}

Equivalence classes for variables: arrays

Eq. 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 variables: compound data type

Arrays in Java and records, or structures, in 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 example illustrates the derivation of equivalence classes for an input variable that has a compound type.

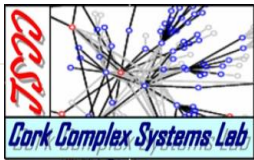
Equivalence classes for variables: compound data type: Example

```
struct transcript
        {
                string fName; // First name.
                string lName; //  Last name.
                string cTitle [200]; // Course titles.
                char grades [200]; // Letter grades corresponding
                                        to course titles.
}
```

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*
- *However, it is incomplete*



Multidimensional partitioning

Multidimensional equivalence partitioning

- Guarantees completeness of testing
- A x B x C given domains A, B, C

- Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually.
- Many classes so created might be infeasible.
- Equivalence classes offer completeness of tests

Partitioning Example

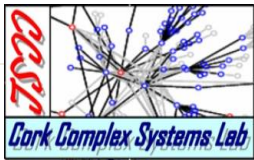
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$.

For unidimensional partitioning we apply the partitioning guidelines to x and y individually. This leads to the following six equivalence classes.

Partitioning Example (contd.)

E1: $x<3$ E2: $3 \leq x \leq 7$ E3: $x>7$ ⟵ y ignored.

E4: $y<5$ E5: $5 \leq y \leq 9$ E6: $y>9$ ⟵ x ignored.



Partitioning Example (contd.)

For multidimensional partitioning we consider the input domain to be the set product $X \times Y$. This leads to 9 equivalence classes.

E1: $x<3, y<5$

E2: $x<3, 5 \leq y \leq 9$

E3: $x<3, y>9$

E4: $3 \leq x \leq 7, y<5$

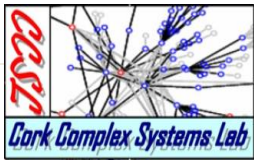
E5: $3 \leq x \leq 7, 5 \leq y \leq 9$

E6: $3 \leq x \leq 7, y>9$

E7: $>7, y<5$

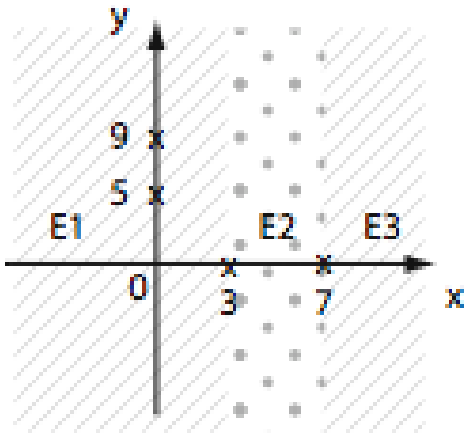
E8: $x>7, 5 \leq y \leq 9$

E9: $x>7, y>9$

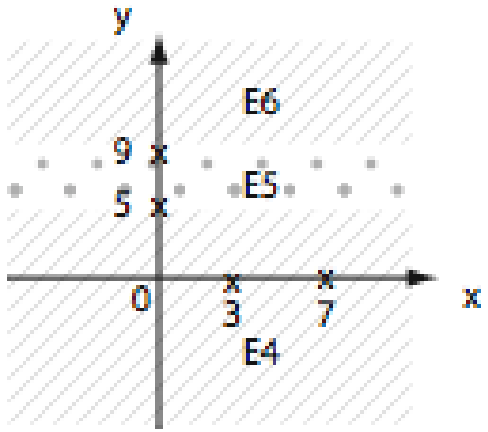


Partitioning Example (contd.)

6 equivalence classes:



(a)



(b)

E1: x<3, y<5

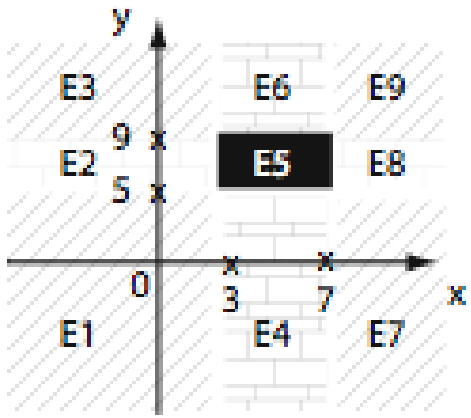
E3: x<3, y>9

E2: x<3, 5≤y≤9

E4: 3≤x≤7, y<5

E5: 3≤x≤7, 5≤y≤9

E6: 3≤x≤7, y>9



(c)

9 equivalence classes:

Systematic procedure for equivalence partitioning

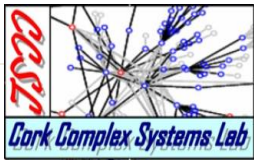
1. Identify the input domain: Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.

Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.

Systematic procedure for equivalence partitioning (contd.)

2. Equivalence classing: Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. partitioning the input domain using values of one variable, is done based on the the expected behavior of the program.

Values for which the program is expected to behave in the ``same way'' are grouped together. Note that ``same way'' needs to be defined by the tester.



Systematic procedure for equivalence partitioning (contd.)

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.

The equivalence classes are combined using the multidimensional partitioning approach described earlier.

Systematic procedure for equivalence partitioning (contd.)

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.

For example, suppose that an application is tested via its GUI, i.e. data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence infeasible.

Boiler control example (BCS)

The control software of BCS, abbreviated as CS, is required to offer several options. One of the options, C (for control), is used by a human operator to give one of four commands (cmd): change the boiler temperature (temp), shut down the boiler (shut), and cancel the request (cancel).

Command temp causes CS to ask the operator to enter the amount by which the temperature is to be changed (tempch).
Values of tempch are in the range -10..10 in increments of 5 degrees Fahrenheit. A temperature change of 0 is not an option.

BCS: example (contd.)

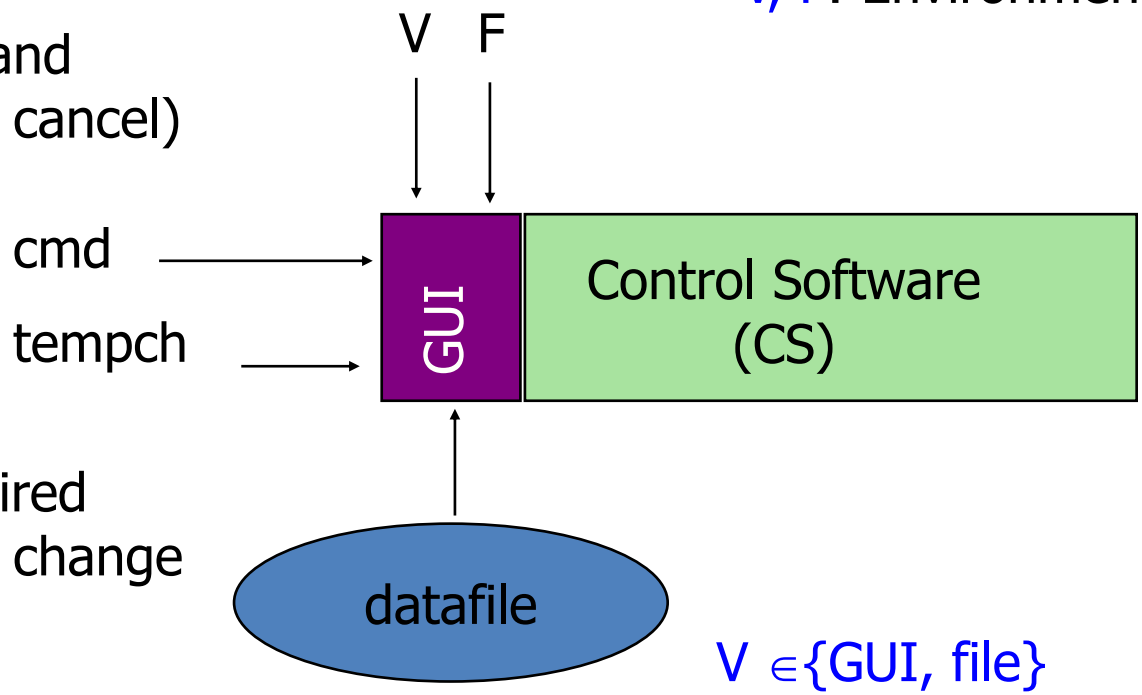
Selection of option C forces the BCS to examine variable V. If V is set to GUI, the operator is asked to enter one of the three commands via a GUI. However, if V is set to file, BCS obtains the command from a command file.

The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is temp. The file name is obtained from variable F.

BCS: example (contd.)

V, F: Environment variables

cmd: command
(temp, shut, cancel)



tempch: desired
temperature change
(-10..10)

V ∈{GUI, file}

F: file name if V is set to "file."

BCS: example (contd.)

Values of V and F can be altered by a different module in BCS. In response to temp and shut commands, the control software is required to generate appropriate signals to be sent to the boiler heating system.

BCS: example (contd.)

We assume that the control software is to be tested in a simulated environment. The tester takes on the role of an operator and interacts with the CS via a GUI.

The GUI forces the tester to select from a limited set of values as specified in the requirements. For example, the only options available for the value of tempch are -10, -5, 5, and 10. We refer to these four values of tempch as tvalid while all other values as tinvalid.

BCS: 1. Identify input domain

The first step in generating equivalence partitions is to identify the (approximate) input domain. Recall that the domain identified in this step will likely be a superset of the complete input domain of the control software.

First we examine the requirements, identify input variables, their types, and values. These are listed in the following table.

BCS: Variables, types, values

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	File, GUI
F	Environment	String	A file name
cmd	Input via GUI/File	Enumerated	{temp, cancel, shut}
tempch	Input via GUI/File	Enumerated	{-10, -5, 5, 10}

BCS: Input domain

Input domain$\subseteq$S=V$\times$F$\times$cmd$\times$tempch

Sample values in the input domain (--: don't care):

(GUI, --, shut, --), (file, cmdfile, shut, --)

(file, cmdfile, temp, 0) ⟵ *Does this belong to the input domain*?

BCS: 2. Equivalence classing

Variable	Partition
V	{{GUI}, {file}, {undefined}}
F	{{fvalid}, {finvalid}}
cmd	{{temp}, {cancel}, {shut}, {cinvalid}}
tempch	{{tvalid}, {tinvalid}}

BCS: 3. Combine equivalence classes (contd.)

Note that tinvalid, tvalid, finvalid, and fvalid denote sets of values. "undefined" denotes one value.

There is a total of $3 \times 4 \times 2 \times 5 = 120$ equivalence classes.

Sample equivalence class: {(GUI, fvalid, temp, -10)}

Note that each of the classes listed above represents an infinite number of input values for the control software. For example, {(GUI}}, fvalid, temp, -10)} denotes an infinite set of values obtained by replacing fvalid by a string that corresponds to the name of an existing file. Each value is a potential input to the BCS.

BCS: 4. Discard infeasible equivalence classes

Note that the GUI requests for the amount by which the boiler temperature is to be changed only when the operator selects temp for cmd. Thus all equivalence classes that match the following template are infeasible.

{(V, F, {cancel, shut, cinvalid}, tvalid$\cup$ tinvalid)}

This parent-child relationship between cmd and tempch renders infeasible a total of $3\times2\times3\times5=90$ equivalence classes.

Exercise: How many additional equivalence classes are infeasible?

BCS: 4. Discard infeasible equivalence classes (contd.)

After having discarded all infeasible equivalence classes, we are left with a total of 18 testable (or feasible) equivalence classes.

Selecting test data

Given a set of equivalence classes that form a partition of the input domain, it is relatively straightforward to select tests. However, complications could arise in the presence of infeasible data and don't care values.

In the most general case, a tester simply selects one test that serves as a representative of each equivalence class.

Exercise: Generate sample tests for BCS from the remaining feasible equivalence classes.

GUI design and equivalence classes

While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must account for the possibility of errors in data entry. For example, the requirement for an application.

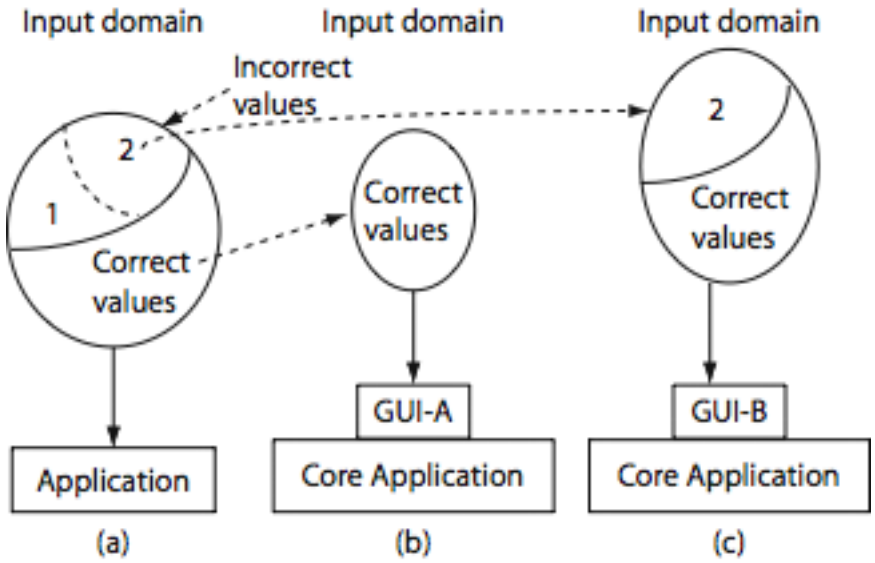
The application places a constraint on an input variable X such that it can assume integral values in the range 0..4. However, testing must account for the possibility that a user may inadvertently enter a value for X that is out of range.

GUI design and equivalence classes (contd.)

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 figure on the next slide.

GUI design and equivalence classes (contd.)



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 f1 when $x \leq 0$ is true and function f2 otherwise. However, M has an error due to which it computes f1 for $x<0$ and f2 otherwise.

Obviously, this fault is revealed, though not necessarily, 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 $x>0$.

Boundary value analysis (BVA)

Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.

While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.

Certainly, tests derived using either of the two techniques may overlap.

BVA: Procedure

1. Partition the input domain using unidimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.

2. Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst the inputs.

3. Select test data such that each boundary value occurs in at least one test input.

BVA: Example: 1.

Create equivalence classes

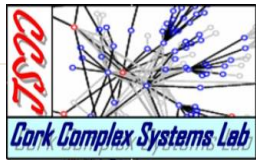
Assuming that an item code must be in the range 99..999 and quantity in the range 1..100,

Equivalence classes for code:

- E1: Values less than 99.
- E2: Values in the range.
- E3: Values greater than 999.

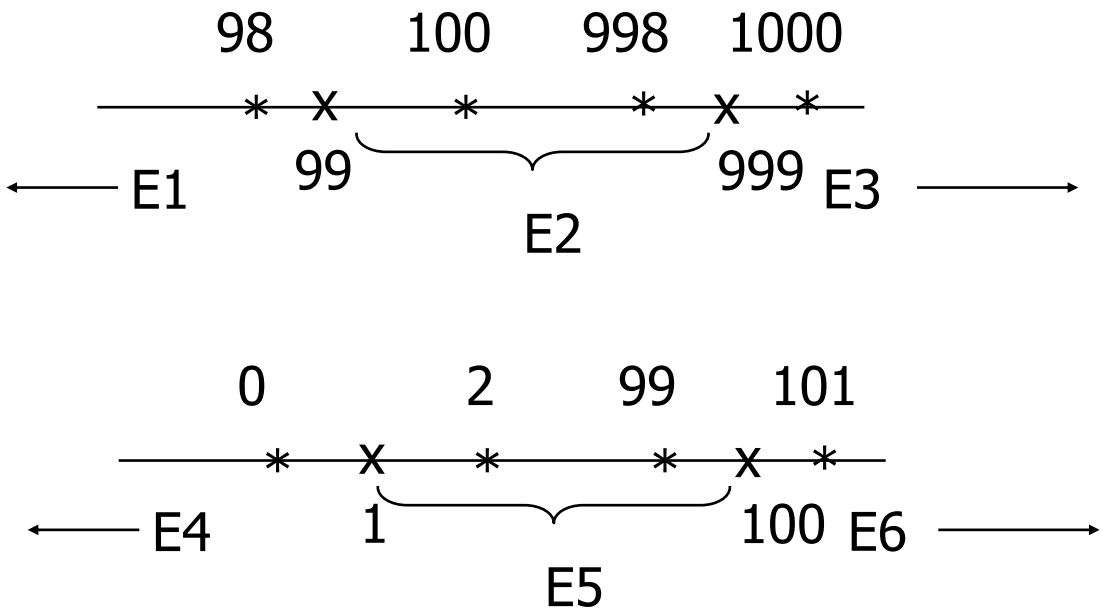
Equivalence classes for qty:

- E4: Values less than 1.
- E5: Values in the range.
- E6: Values greater than 100.



BVA: Example:

2. Identify boundaries



Equivalence classes and boundaries for findPrice. Boundaries are indicated with an x. Points near the boundary are marked *.

BVA: Example:

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. Consider the following test set:

T={ t1: (code=98, qty=0),
t2: (code=99, qty=1),
t3: (code=100, qty=2),
t4: (code=998, qty=99),
t5: (code=999, qty=100),
t6: (code=1000, qty=101)
}

Illegal values of code and qty included.

BVA: Recommendations

Relationships amongst the input variables must be examined carefully while identifying boundaries along the input domain. This examination may lead to boundaries that are not evident from equivalence classes obtained from the input and output variables.

Additional tests may be obtained when using a partition of the input domain obtained by taking the product of equivalence classes created using individual variables.

Summary

Equivalence partitioning and boundary value analysis are the most commonly used methods for test generation while doing functional testing.

Given a function *f* to be tested in an application, one can apply these techniques to generate tests for *f*.