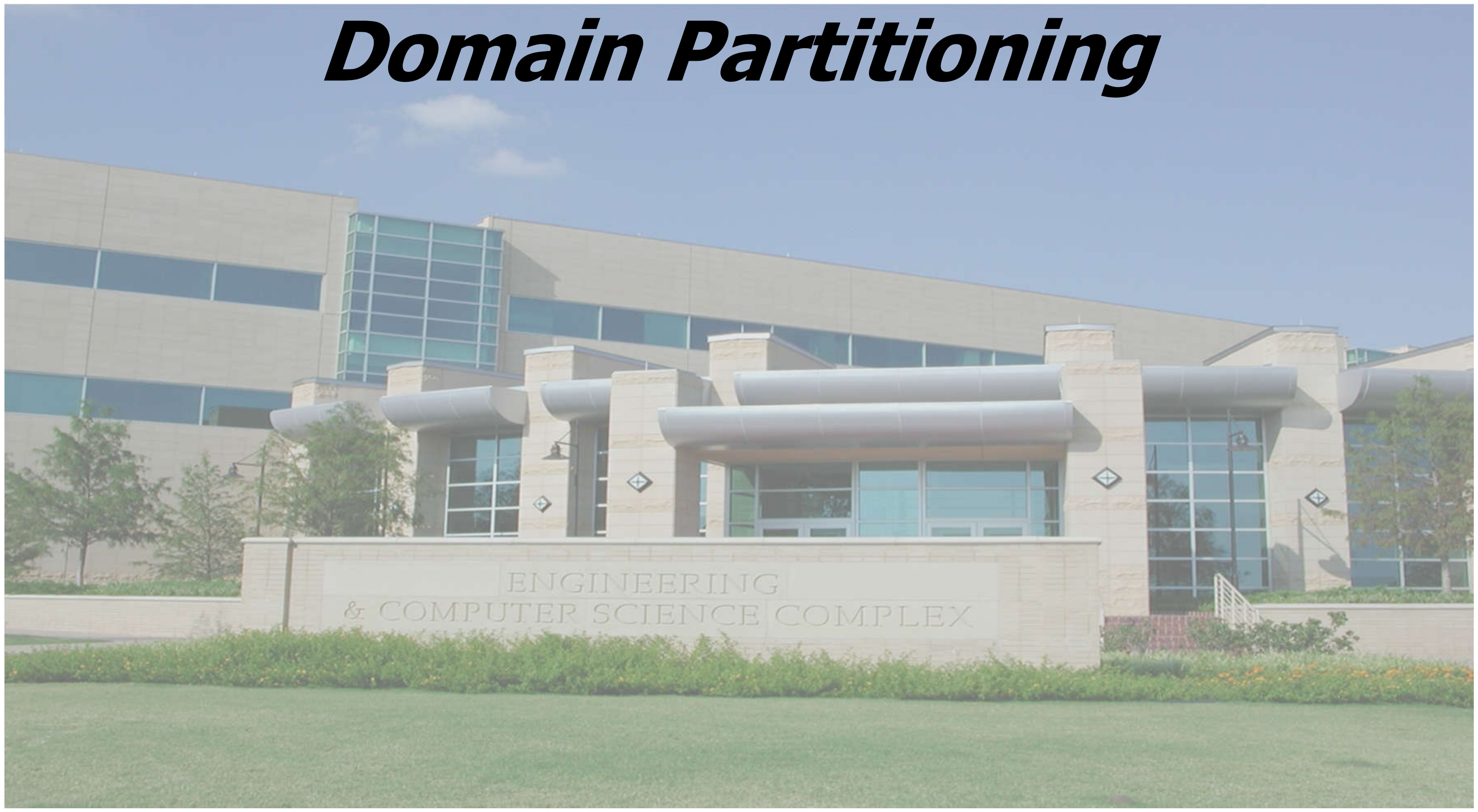
Domain Partitioning



Dr. Mark C. Paulk

SE 4367 – Software Testing, Verification, Validation, and Quality Assurance

Topics: Software Testing

<u>Part II: Test Generation</u>

→ **3. Domain Partitioning**
- **Equivalence Partitioning**
- **Boundary Value Analysis**
- **Category-Partition Method**

4. Predicate Analysis

5. Test Generation from Finite State Models

6. Test Generation from Combinatorial Designs

Requirements Are the Starting Point

Requirements specifications may be informal, rigorous, and/or formal.

The input domain is derived from the informal and rigorous specifications.

Black-box testing…

Test Selection Problem

Select a subset T of test such that execution of program p against each element of T will reveal all errors in p.

In general, there does not exist an algorithm to construct such a test set.

The problem of test selection is primarily because of the size and complexity of the input domain of p.

Mathur, Example 3.1

P sorts a sequence of integers into ascending order.

- **input domain of integers [-32768, 32767]**
- **limit $N_{max} > 1$**
- **then the size of the input domain depends on the value of N**

S, the size of the input domain, is given by the formula where *v* is the number of possible values each element of the input sequence may assume, i.e., 65,536.

$$S = \sum_{i=1}^{N} \upsilon^{i}$$

Equivalence Partitioning

Subdivide the input domain into a relatively small number of subdomains.

Partition → subdomains are disjoint

Each subset is known as an equivalence class.

Assumes P exhibits the same behavior for every element within the class.

E = legal inputs
U = illegal (unexpected) inputs

Mathur, Example 3.5

wordCount

- **takes a word w and a filename f as input**
- **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 legal / illegal (E and U) partitioning method, there are two equivalence classes.

- **black-box**

E1: (w,f) where w is a string and f is an existing file

E2: (w,f) where w is a string and f is a file that does not exist

Program P2.1 (P3.1)

```
1) begin
2) string w, f;
3) input (w, f);
4) if (not exists(f)) {
       raise exception;
       return(0);
       }
5) if (length(w)==0) return(0);
6) if (empty(f)) return(0);
7)   return(getCount(w,f));
8) end
```

How many paths?

- **8**

How many feasible paths?

- **4 (Mathur says 6)**
- **each if-then terminates (return) the program**

How many equivalence classes?

- **4 (Mathur says 6)**
- **depending on which of the four feasible paths is covered by a test case**

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

- does not matter whether w is null if file does not exist

~~E6~~	~~null~~	~~exists, empty~~

- does not matter whether file is empty if word is null

Using Outputs

We could base equivalence classes on the outputs generated by a program.

Derive equivalence classes for the inputs based on the output equivalence classes.

Only if analyzing the inputs and requirements is insufficient.

Equivalence Classes for Variables

Range (integers and floating point)

- **one class with values inside the range**
  - **there may be multiple legal ranges, which may or may not be adjacent**
- **two classes with values outside the range**
  - **less than**
  - **greater than**

Strings

- **at least one class containing legal strings**
- **at least one class containing illegal strings**
- **the empty string ε is an (illegal) equivalence class**
- **legality is determined based on constraints on the length and other semantic features of the string**

Enumerations

- **each value in a separate class**
- **may combine values if program behavior is the same**

Arrays

- **one class containing all legal arrays**
  - **may be additional legal equivalence classes depending on semantics, e.g., values must be in [-3,3]**
- **one class containing only the empty array**
- **one class containing arrays larger than the expected size**

Compound data types (e.g., structures in C++)

- **legal and illegal values for each component**

String ECP Example

The input is a string that is 8 alphabetic characters long.

- **The first character is either "R" or "W".**
- **Characters 2-8 are in {A-Z}.**

E_1: "Rxxxxxxx" – legal input

E_2: "Wxxxxxxx" – legal input

- **where x is in {A-Z}**

E_3: "Rxxxx" – illegal input, short R string

E_4: "Wxxxx" – illegal input, short W string

E_5: "Xxxxxxxx" – illegal input, not R/W

- **where X is not R or W**

E_6: "Rxxxxxxxxxx" – illegal input, long R string

E_7: "Wxxxxxxxxxx" – illegal input, long W string

E_8: "Rxxx&xxx" – illegal input, non-A-Z R string

E_9: "Wxxx&xxx" – illegal input, non-A-Z W string

- **where & is not an alphabetic character**

E_{10}: ε – illegal input, null string

Questions: String ECP Example

Applying Judgment and Knowledge

Should you have equivalence classes with more than one mistake?

Do you need "short" strings for both R and W?

- **how long should a short string be?**
- **1 character (R/W)?**
- **2 characters?**
- **7 characters?**

Do you need "long" strings for both R and W?

- **how long should a long string be?**
- **9 characters?**

Does it matter what position the non-alphabetic characters are in?

Unidimensional Partitioning

One input variable at a time

- **simple and scalable**

Multidimensional partitioning can become too large.

Combining illegal inputs does not add much value.

- **Addressing one illegal input frequently masks subsequent illegal inputs.**
- **Combinations of errors can cause unexpected results, so it's a judgment call.**

Mathur, Example 3.7

Application with integer inputs x and y

- **$3 \leq x \leq 7$**
- **$5 \leq y \leq 9$**

Unidimensional partitioning

E_1: $x < 3$	E_2: $3 \leq x \leq 7$	E_3: $x > 7$
E_4: $y < 5$	E_5: $5 \leq y \leq 9$	E_6: $y > 9$

Multidimensional partitioning

E_1: $x < 3$, $y < 5$	E_2: $x < 3$, $5 \leq y \leq 9$	E_3: $x < 3$, $y > 9$
E_4: $3 \leq x \leq 7$, $y < 5$	E_5: $3 \leq x \leq 7$, $5 \leq y \leq 9$	E_6: $3 \leq x \leq 7$, $y > 9$
E_7: $x > 7$, $y < 5$	E_8: $x > 7$, $5 \leq y \leq 9$	E_9: $x > 7$, $y > 9$

Procedure for Equivalence Partitioning

Identify the input domain

Equivalence classing

- **partition the input domain**
- **tester defines *same way* of program behavior**
- **consider output-driven equivalence classes**

Combine equivalence classes

- **usually omitted (e.g., enumerated variables)**
- **multidimensional partitioning**

Identify infeasible equivalence classes

- **GUI interface may only allow valid inputs**

GUI Design

GUI may offer only correct choices via menu.

GUI may ask the user to fill in a box.

- **illegal values are possible**

Test design must take into account GUI design.

Makes the assumption the GUI has been correctly implemented.

Boundary Value Analysis

Programmers make mistakes in process values at or near the boundaries of equivalence classes.

The "requirement" is

```
        if (x ≤ 0) then return f1; else return f2;
```

But the program executes

```
        if (x < 0) then return f1; else return f2;
```

x=0 lies at the boundary between the equivalence classes…

As a Test Selection Technique

Focuses on tests at and near the boundaries of equivalence classes

Recommended that boundaries be identified based on the relations among the input variables

Boundary Value Analysis Procedure

Partition the input domain using unidimensional partitioning.

- **as many partitions as there are input variables**
  - **a single partition of an input domain can also be created using multidimensional partitioning**

Identify the boundaries for each partition.

  - **also use any special relationships among inputs**

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

- **near-boundary values: just inside, just outside**

Mathur, Example 3.11

findPrice application

- ***code*** **in [99, 999]**
- ***quantity*** **in [1, 100]**

Equivalence classes for *code*

- **E1: *code* < 99**
- **E2: 99 ≤ *code* ≤ 999**
- **E3: code > 999**

Equivalence classes for quantity

- **E4: *quantity* < 1**
- **E5: 1 ≤ *quantity* ≤ 100**
- **E6: *quantity* > 100**

Boundaries for *code*: 99, 999

***code* values near the boundaries: 98, 100, 998, 1000**

Boundaries for *quantity*: 1, 100

***quantity* values near the boundaries: 0, 2, 99, 101**

A test set that includes all values at or near the boundaries…

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)
}

T is a minimal set of tests that includes all boundary and near boundary values.

Is T a “good” test set?

There is no test case for a legal value of *code* and an illegal value of *quantity*…

Could replace t_1 and t_6 with

- **t_7: (*code*=98, *quantity*=45)**
- **t_8: (*code*=1000, *quantity*=45)**
- **t_9: (*code*=250, *quantity*=0)**
- **t_{10}: (*code*=250, *quantity*=101)**

Testing Advice from Kaner

If you expect the same result from two tests, use only one of them.

- **equivalence class partitioning**

When you choose representatives of a class for testing, always pick the ones you think the program is most likely to fail.

- **The best cases are at the boundaries of a class.**
  - **boundary value analysis**
- **Not every boundary in a program is intentional, and not all intended boundaries are set correctly.**
  - **largest value that will fit in a computer word: input and output**
  - **negative numbers as well as positive**
  - **zero**

Category-Partition Methods

A systematic approach to generation of tests from requirements

- **a systematization of equivalence partitioning and boundary value techniques**

Mix of manual and automated steps

Transforms requirements into test specifications

- **categories corresponding to program inputs and environment objects**

Test specs are input to test-frame generator

- **generate test scripts**

Test Frames

A collection of choices, corresponding to each category

A template for one or more test cases that are combined into one or more test scripts

Steps in the Category-Partition Method

- Functional specification

1) Analyze specification
    - Functional units
2) Identify categories
    - Categories
3) Partition categories
    - Choices
4) Identify constraints
    - Constraints
5) (Re)write test specification
    - Test specification (TS)
6) Process specification
    - Test frames
7) Evaluate generator output: revise TS, goto step 6
    - Test frames
8) Generate test scripts
    - Test scripts

Mathur Running Example for Category-Partition Method

findPrice application

- **8-digit *code***
- ***qty***
- ***weight***

***code* leftmost digit**	**Interpretation**
0	**ordinary grocery items**
2	**variable-weight items**
3	**health-related items**
5	**coupon**
1, 6-9	**unused**

fP finds unit price, description, total price of item

- **displays error message if any of three is incorrect**

Running Example
Keying on "code"

Use of *qty* and *weight* depends on leftmost digit of *code*

• **0 or 3**	***qty*= quantity purchased, integer** ***weight* = not used**
• **2**	***qty* = not used** ***weight* = weight of item**
• **5**	**second digit is $ amount** **third and fourth digits are ¢ amount**
• **1, 6-9**	**ignored**

- **what about 4?**

Running Example
1) Analyze Specification

Identify each functional unit to test separately

findPrice → fP

Running Example
2) Identify Categories

Isolate inputs, identify objects in environment

Determine characteristics (categories)

- ***qty* and *weight* are related to *code***
- **no bounds on *qty* and *weight***
- **fP accesses database → environment object**

***code*: length, leftmost digit, remaining digits**
***qty*: integer**
***weight*: float**
database: contents???

Running Example
3) Partition Categories

What are the different cases (choices) against which the functional unit must be tested?

Partition each category into at least two subsets: correct values + erroneous values

- **in a networked environment, this might include events like network failure**

Running Example
Inputs and Environment Objects

code

- length
  - valid (8 digits)
  - invalid (<,> 8 digits)
- leftmost digit
  - 0
  - 2
  - 3
  - 5
  - others
- remaining digits
  - valid string
  - invalid string???

qty

- integer
  - valid quantity
  - invalid quantity (0)

weight

- float
  - valid weight
  - invalid weight (0)

Is there an upper bound to qty or weight?

Environment: database

- item exists
- item does not exist

Running Example
4) Identify Constraints

A test for a functional unit consists of a combination of choices for each parameter and environment object.

- **Certain combinations might not be possible.**
- **Some combinations must satisfy specific relationships.**

A constraint is specified using a property list and selector expression.

[property   P1, P2, …]

- **property is a key word**
- **P1, P2, … are the names of individual properties**

Running Example
Selector Expressions

[if P]

[if P1 and P2 and …]

[error]

- **can be assigned to error conditions**

[single]

- **specifies that the associated choice is not to be combined with choices of other parameters or environment objects**

Running Example
Sampled Choices

Comment lines start with #

```
# Leftmost digit of code
0       [property ordinary-grocery]
2       [property variable-weight]
# Remaining digits of code
valid string           [single]
# Valid value of qty
valid quantity         [if ordinary-grocery]
# Incorrect value of qty
invalid quantity       [error]
```

Running Example
5) (Re)write Test Specification

Write test spec in a test specification language (TSL)

```
Parameters
code
    length
        valid
        invalid          [error]
leftmost digit
0       [property Ordinary-grocery]
2       [property Variable-weight]
3       [property Health-related]
5       [property Coupon]
```

remaining digits	
valid string	**[single]**
invalid string	**[error]**
qty	
valid quantity	**[if Ordinary-grocery]**
invalid quantity	**[error]**
weight	
valid weight	**[if Variable-weight]**
invalid weight	**[error]**
Environments: database	
item exists	
item does not exist	**[error]**

Running Example
6) Process Specification

TSL specification is processed by an automatic test-frame generator

Analyze for redundancy

Sample test frame:

Test number identifies the test

Test case 2: (Key=1.2.1.0.1.1)

Key identifies the choices
0 indicates no choice

Length:	**valid**
Leftmost digit:	**2**
Remaining digits:	**valid string**
***qty*:**	**ignored**
***weight*:**	**3.19**
database:	**item exists**

Running Example
Test Frames

A test frame is not a test case.

- **test cases are derived from test frames**

Test frames are generated from all possible combinations of choices while satisfying the constraints.

Choices marked [error] or [single] are not combined with others and only produce one test case.

Running Example
7) Evaluate Generator Output

Examine test frames for redundancy or missing cases

Think about a code 4 input…

Running Example
8) Generate Test Scripts

Combine test cases generated from test frames into test scripts

A test script is a grouping of test cases

- **group test cases that do not require any changes in settings of the environment objects**

Summary – Things to Remember

Equivalence class partitioning

Boundary value analysis

Questions and Answers