3. Test Generation Strategies II

Based on requirements

Andrea Polini

Software Engineering II – Software Testing
MSc in Computer Science
University of Camerino

November 4th, 2014

Equivalence classes for variables

There are some guidelines to define equivalence classes on the base of variables domains and defined requirements. They reflect possible implementation choices related to explicit knowledge or implicit one:

- Range (implicitly or explicitly defined)
- Strings
- Enumerations
- Arrays
- Compound Data Types

Unidimensional vs. Multidimensional partitioning

Unidimensional

Each input variable is considered per-se and classes are combined to cover all the possible equivalence classes

Multidimensional

The Cartesian product of equivalence classes is considered and test derived accordingly.

Systematic procedure

- Identify input domains
- Equivalence classing
- Combine equivalence classes
- Identify infeasible equivalence classes

The Boiler Control System (BCS)

BCS

The control system takes in input:

- A command: $cmd = (temp|shut|cancel)$
- When *temp* is selected $tempch = -10|-5|5|10$

Input can be provided via a GUI or via a configuration file.

How would you partition the input domain?

BCS input domain

Variable	Type	Value(s)
V	Enumerated	$\{GUI, file\}$
F	String	*A file name*
cmd	Enumerated	$\{temp, cancel, shut\}$
$tempch$	Enumerated	$\{-10, -5, 5, 10\}$

The Boiler Control System (BCS)

BCS

The control system takes in input:

- A command: $cmd = (temp|shut|cancel)$
- When *temp* is selected $tempch = -10|-5|5|10$

Input can be provided via a GUI or via a configuration file.

How would you partition the input domain?

BCS input domain

Variable	**Type**	**Value(s)**
V	Enumerated	$\{GUI, file\}$
F	String	*A file name*
cmd	Enumerated	$\{temp, cancel, shut\}$
$tempch$	Enumerated	$\{-10, -5, 5, 10\}$

Boundary-value analysis

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

Boundary-value analysis

test-selection techniques that targets faults in applications at the boundaries of equivalence classes.

Once the input domain has been identified:

- Partition the input domain using unidimensional partitioning
- Identify the boundaries of each partition
- Select test data such that each boundary value occurs in at least one test input

Boundary-value analysis

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

Boundary-value analysis

test-selection techniques that targets faults in applications at the boundaries of equivalence classes.

Once the input domain has been identified:

- Partition the input domain using unidimensional partitioning
- Identify the boundaries of each partition
- Select test data such that each boundary value occurs in at least one test input

Boundary-value analysis

Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes

Boundary-value analysis

test-selection techniques that targets faults in applications at the boundaries of equivalence classes.

Once the input domain has been identified:

- Partition the input domain using unidimensional partitioning
- Identify the boundaries of each partition
- Select test data such that each boundary value occurs in at least one test input

BA Example

The *findPrice* procedure

Two integer parameter *code* and *quantity* with the following input domain:

- $99 \leq code \leq 999$
- $1 \leq quantity \leq 100$

Which are the relevant partitions?

Which are the relevant boundary values?

BA Example

The *findPrice* procedure

Two integer parameter *code* and *quantity* with the following input domain:

- $99 \leq code \leq 999$
- $1 \leq quantity \leq 100$

Which are the relevant partitions?
Which are the relevant boundary values?

BA Example

Consider the following test set:

$$
\begin{aligned}
T = \{ \quad & t_1 : (code = 98, quantity = 0), \\
& t_2 : (code = 99, quantity = 1), \\
& t_3 : (code = 100, quantity = 2), \\
& t_4 : (code = 998, quantity = 99), \\
& t_5 : (code = 999, quantity = 100), \\
& t_6 : (code = 1000, quantity = 101), \\
\} &
\end{aligned}
$$

Minimal but:

```
public void fP(int code, int quantity) {
  if (code < 99 && code > 999)
    {display_error("invalid code"); return;}
    // Validity check for quantity is missing!
    // Begin processing code and quantity
  ...
}
```

BA Example

Consider the following test set:

$$
\begin{aligned}
T = \{ \quad & t_1 : (code = 98, quantity = 0), \\
& t_2 : (code = 99, quantity = 1), \\
& t_3 : (code = 100, quantity = 2), \\
& t_4 : (code = 998, quantity = 99), \\
& t_5 : (code = 999, quantity = 100), \\
& t_6 : (code = 1000, quantity = 101), \\
\} &
\end{aligned}
$$

Minimal but:

```
public void fP(int code, int quantity) {
  if (code < 99 && code > 999)
    {display_error("invalid code"); return;}
    // Validity check for quantity is missing!
    // Begin processing code and quantity
  ...
}
```

Category Partition Method

the findPrice procedure (2nd version)

$$findPrice(code, quantity, weight)$$

Leftmost digit	Interpretation
0	Ordinary grocery items such as bread, magazines soup
2	Variable-weight items such as meats, fruits, and vegetables
3	Health-related items such as tylenol, bandaids, and tampons
5	Coupon; digit 2(dollars), 3 and 4 (cents) specify the discounts
1, 6-9	unused

Category Partition Method

CP Method

Mixed manual/automatic approach consisting of eight successive steps based approach to go from requirements to test scripts

- Analyze specification
- Identify Categories
- Partition Categories
- Identify Constraints
- (Re)write test specification
- Process Specification
- Evaluate Generator Output
- Generate Test Scripts

Analyze Specification

The tester identify each functional unit that can be tested separately

Identify categories

For each testable unit spec analyzed and inputs isolated. Also objects in the environment are considered. Then the relevant characteristics (category) of each parameter are identified

findPrice

Categories:

- *code*: length, leftmost digits, remaining digits
- *quantity*: integer quantity
- *weight*: float quantity
- *database*: contents

Partition Categories

For each category different cases (choices) against which to test the functional units are identified.

- *code*:
  - Length: Valid (8 digits), Invalid ($<$ or $>$ 8)
  - leftmost digit: 0,2,3,5,others
  - remaining digits: valid string, invalid string
- *quantity*: valid, invalid
- *weight*: valid, invalid
- *Database*: item exists, item does not exist

Identify Constraints

Constraints among choices are specified and used to exclude infeasible tests

(Re)write test specification

The tester write a complete test specification using a TSL, and taking into account the information derived in the previous steps

Process Specification

TSL specification are processed top derive test frames.

Evaluate Generator Output

Generated tests are analyzed for redundancy of missing cases

Generate Test Scripts

Test frames are finally grouped into test scripts

CP is mainly a systematization of the equivalence partitioning and boundary value analysis

Generate Test Scripts

Test frames are finally grouped into test scripts

CP is mainly a systematization of the equivalence partitioning and boundary value analysis

Cause Effect Graphing