Test Generation from Requirements

Lecture 4

Dr. Ilchul Yoon (icyoon@sung.korea.ac.kr)

Sources: (1) Foundations of Software Testing (textbook), (2) slides by Prof. Mathur

Requirement Modeling Formalisms

- More formal, higher chances of automated test generation
  - Easier to generate test cases with FSM or Petri net, compared to use cases

- Different formalisms for different software property
  - Petri nets for timing and concurrency properties in distributed system
  - Time input automata for timing constraints in a real-time system
  - FSMs for capturing state transitions in a protocol
  - UML – a combination of multiple notations

Test-Selection Problem

- Reality
  - p's input domain is huge and complexity is high.

- Mission
  - Select a subset $T$ of tests such that execution of a program $p$ against each element of $T$ will reveal all errors in $p$

    No such algorithm exists!

- Heuristics and model-based methods can be used.

- Question
  - How to construct the test set $T \subseteq D$ (input domain) that will reveal as many errors in $p$ as possible?
Test Input Domain is Large!

- Consider a procedure P in a payroll-processing system.
  - Input: employee’s record
  - Output: weekly salary per employee

- Value ranges of each element in the input domain
  - ID: int: 10 is three-digit long from 001 to 999.
  - name: string: name is 20-character 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: hours worked vary from 0 to 10.

- Number of all possible values
  \[ 999 \times 27^{20} \times 21 \times 61 \approx 5.42 \times 10^{54}. \]

Equivalent Partitioning

- The idea
  - Subdivide input domain into multiple disjoint subdomains (equivalence classes)
  - Program would exhibit identical behavior on values in a subdomain.
  - Simply choose one test from each domain.

- Observations
  - Different partitions are possible
  - Fault detection power depends on the capability of testers.

Equivalence Partitioning

- Select tests targeting any faults
- An example - Health care support system
  - Partition E: Legal input value range: [1-120]
  - Partition U: Illegal input value range: all other integers
  - E can be further divided.
    - Can be claimed as a dependent: [1-26]
    - Medicare is eligible: [65-120]
    - Other private insurance: [27-64]
  - U can also be further divided.
    - Values less than 1 vs. values greater than 120

Measuring the effectiveness

- Ratio of the number of faults exposed by the tests to the total faults lurking in A

Relations and Equivalence Partitioning

- Relations for domain partitioning are of the form: \( R: \mathcal{I} \rightarrow \mathcal{I} \).
  - \( R \) defines equivalence classes that are subsets of \( \mathcal{I} \).
- Partition based on the knowledge of requirements
- Example 1

- Example 3: Consider a method pPrice that takes the name of a grocery item as input, consults a database of prices, and returns the unit price of this item. If the item is not found in the database, it returns with a message Price information not available.

- pFound relation associates two elements
  - in an equivalence class if the prices for both elements are found
  - in another class if the prices are not found for both
Example 2

Example 2: Consider an automatic printer testing application named pTest. The application takes the manufacturer and model of a printer as input and selects a test script from a list. The script is then executed to test the printer. Our goal is to test if the script selection part of the application is implemented correctly. Assume 4 different printer types (ci, cl, cm, invP) for this example.

- Approach #1
  - Use 4 different relations
  - Generates 8 different non-disjoint classes

- Approach #2
  - Use 1 relation that partitions the domain into 4 eq. classes

\[ p\text{Cat}: \text{I} \rightarrow \text{I} \]

Multiple different equivalence relations are possible!

Example (cont.)

Suppose that the tester can access the code for wordCount.

Program PE1:

1 begin
2 string w, f;
3 input (w, f);
4 if (-exists(f)) {raise exception; return(0)};
5 if (length(w)==0) return(0);
6 if (empty(f)) return(0);
7 return(wordCount(w, f));
8 end

Can define relations that partitions to 6 classes

<table>
<thead>
<tr>
<th>Equivalence class</th>
<th>w</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>non-null</td>
<td>exists, nonempty</td>
</tr>
<tr>
<td>E2</td>
<td>non-null</td>
<td>does not exist</td>
</tr>
<tr>
<td>E3</td>
<td>non-null</td>
<td>exists, empty</td>
</tr>
<tr>
<td>E4</td>
<td>null</td>
<td>exists, nonempty</td>
</tr>
<tr>
<td>E5</td>
<td>null</td>
<td>does not exist</td>
</tr>
<tr>
<td>E6</td>
<td>null</td>
<td>exists, empty</td>
</tr>
</tbody>
</table>

Relations and Equivalence Partitioning

- Partition based on the knowledge of requirements and the program text

Example

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.

1. E1: Consists of pairs (w, f), where w is a string and f denotes a file that exists.
2. E2: Consists of pairs (w, f), where w is a string and f denotes a file that does not exist.

It is worth asking: “Does the program ever generate a 0? What are the maximum and minimum possible values of the output?”

4 classes from the 2 questions:

- 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.
### Equivalence Classes for Variables: Range

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed ( [60..90] )</td>
<td>{(50), (75), (92)}</td>
</tr>
<tr>
<td>area: float ( &gt;8.0 )</td>
<td>{{-1.0}, {15.52}}</td>
</tr>
<tr>
<td>age: int</td>
<td>{{-1}, {56}, {132}}</td>
</tr>
<tr>
<td>letter: bool</td>
<td>{{}, {3}}</td>
</tr>
</tbody>
</table>

One class with values inside the range and two with values outside the range.

### Equivalence Classes for Variables: String

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstname: string</td>
<td>{{e}, {Sue}, {Looong Name}}</td>
</tr>
</tbody>
</table>

At least one containing all legal strings and one all illegal strings based on any constraints.

### Equivalence Classes for Variables: Enumeration

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocolor: {red, blue, green}</td>
<td>{{red}, {blue}, {green}}</td>
</tr>
<tr>
<td>up: boolean</td>
<td>{{true}, {false}}</td>
</tr>
</tbody>
</table>

Each value in a separate class.

### Equivalence Classes for Variables: Arrays

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int [] aName: new int[3];</td>
<td>{{}, {{-10, 20}}, {{-9, 0, 12, 15}}}</td>
</tr>
</tbody>
</table>

One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.
Eq. Classes for Vars: Compound Data Types

- While generating equivalence classes for compound data types, consider legal and illegal values for each component of the structure.
- Example

```c
struct S {
  1 string fName; // First name.
  2 string lName; // Last name.
  3 string cTitle; // Course title.
  4 char grades[200]; // Letter grades corresponding to course titles.
};
```
- Derive equivalence classes for each component of `R` and combine them!

Unidimensional Partitioning

- Create one partition for each input variable at a time
- Leads to a partition per each variable where each contains multiple classes.
- Generates fewer test cases
- Commonly used
- Example

Example 8.7: 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 \(5 \leq y \leq 9\). 

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E1:</td>
<td>(x &lt; 3)</td>
<td>E2:</td>
<td>(3 \leq x \leq 7)</td>
</tr>
<tr>
<td>E3:</td>
<td>(x &gt; 7)</td>
<td>E4:</td>
<td>(y &lt; 5)</td>
</tr>
<tr>
<td>E5:</td>
<td>(y = 5)</td>
<td>E6:</td>
<td>(y &gt; 9)</td>
</tr>
<tr>
<td>E7:</td>
<td>(y = 9)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multidimensional Partitioning

- Creates one partition consisting of several equivalence classes
- Leads to a large number of equivalence classes
- Automation would be needed.
- Some classes might be infeasible.
- Offer an increased variety of tests
- Example

Example 8.7: 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 \(5 \leq y \leq 9\). 

Unidimensional vs. Multidimensional

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td></td>
</tr>
</tbody>
</table>

```c
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: x > 7, y < 5
E8: x > 7, 5 \leq y \leq 9
E9: x > 7, y > 9
```
Systematic Procedure for Eq. Partitioning

- Identify the input domain
  - Based on careful requirement review
  - Identify all input and output variables, their types, and any conditions associated
    - Environment variables can also be considered.

- Equivalence classing
  - Partition the set of values of each variable into disjoint subsets. (each subset is an equivalence class.)
  - Values for which the program is expected to show identical behavior are grouped together.

Boiler Control Example (BCS)

- Requirements
  - 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): (1) change the boiler temperature (temp), (2) shut down the boiler (shut), and (3) 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.

- Requirements (cont.)
  - 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.
Boiler Control Example (BCS)

- **cmd**: command (temp, shut, cancel)
- **tempch**: desired temperature change (-10..10)
- **V, F**: Environment variables
- **V**: \{GUI, file\}
- **F**: file name if V is set to “file.”

**Assumption by tester**
- 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 valid while all other values as invalid.

**BCS: 1. Identify input domain**
- Identify input variables, their types, and values.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Kind</th>
<th>Type</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Environment</td>
<td>Enumerated</td>
<td>File, GUI</td>
</tr>
<tr>
<td>F</td>
<td>Environment</td>
<td>String</td>
<td>A file name</td>
</tr>
<tr>
<td>cmd</td>
<td>Input via GUI/File</td>
<td>Enumerated</td>
<td>{temp, cancel, shut}</td>
</tr>
<tr>
<td>tempch</td>
<td>Input via GUI/File</td>
<td>Enumerated</td>
<td>{-10, -5, 5, 10}</td>
</tr>
</tbody>
</table>

**True input domain**

\[ I \subseteq S = V \times F \times cmd \times tempch \]

Sample values in the input domain:
- (GUI, --, shut, --), (file, cmdfile, shut, --)

Does (file, cmdfile, temp, 0) belong to the input domain?
BCS: 2. Equivalence Classing

- **Equivalence classes**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>{GUI}, {file}, {undefined}</td>
</tr>
<tr>
<td>F</td>
<td>f_valid, f_invalid</td>
</tr>
<tr>
<td>cmd</td>
<td>{temp}, {cancel}, {shut}, {c_invalid}</td>
</tr>
<tr>
<td>tempch</td>
<td>t_valid, t_invalid</td>
</tr>
</tbody>
</table>

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

  - Example: \{(GUI, temp, f_valid, -10)\}

    - This example denotes an infinite set of values obtained by replacing f_valid 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 Eq. Classes

- Observations from requirements
  - The GUI requests for the amount by which the boiler temperature is to be changed only when the operator selects temp for cmd.
  - Parent-child relationship between cmd and tempch
  - All equivalence classes that match the following template are infeasible.
    - \{(V, F, \{cancel, shut, cinvalid\}, tvalid \cup tinvalid)\}

    - Makes a total of $3 \times 2 \times 3 \times 5 = 90$ equivalence classes infeasible

    - After having discarded all infeasible eq. classes, only 18 classes remain.