Requirements-based Test Generation

Professor Mark Gabel
Slides courtesy of Professor W. Eric Wong
Department of Computer Science
University of Texas at Dallas

Learning Objectives

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

Essential black box-based techniques for test generation.
Test generation techniques described in this unit belong to the black-box testing category.

These techniques are useful during functional testing where the objective is to test whether or not an application, unit, system, or subsystem, correctly implements the functionality as per the given requirements.

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.*

---

Sample Use Case (1)
Sample Use Case (2)

- Name of Use Case
- Actors
  - Description of actors involved in use case
- Entry condition
  - Use a syntactic phrase such as “This use case starts when…”
- Flow of Events
  - Free form, informal natural language
- Exit condition
  - Star with “This use cases terminates when…”
- Exceptions
  - Describe what happens if things go wrong
- Special Requirements
  - List nonfunctional requirements and constraints

Sample Use Case (3)

| Flow of events  | 1. The Field Officer activates the "Report Emergency" function of her terminal.  
| Participating actor | 2. The Field Officer responds by filling out a form in the Field Officer.  
|                     | 3. The Field Officer completes the form by selecting the emergency level, type, location, and brief description of the situation. The Field Officer also describes possible responses to the emergency situation. Once the form is completed, the Field Officer submits the form.  
|                     | 4. The Field Officer receives the form and notifies the Dispatcher.  
|                     | 5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the ReportIncident use case. The Dispatcher selects a response and acknowledges the report.  
|                     | 6. The Field Officer displays the acknowledgment and the selected response to the Field Officer.  
| Entry condition | The Field Officer is logged in.  
| Exit conditions | The Field Officer has received an acknowledgment and the selected response from the Dispatcher.  
| Quality requirements | The Field Officer’s report is acknowledged within 30 seconds.  
|                     | The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.  

Figure 4.1: An example of a use case, ReportEmergency. Under ReportEmergency, the left column lists interactions, and the right column denotes system responses.
Let $D$ denote the input domain of a program $P$. Ideally, the test selection problem is to select a subset $T$ (of $D$), “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.

The following two examples illustrate what is responsible for large and complex input domains.
Large Input Domain

Consider program $P$ that is required to \textit{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 $P$ consists of \textit{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 \textit{infinitely large} and $P$ can never be tested exhaustively. If the size of the input sequence is limited to, say $N_{\text{max}} > 1$, then the size of the input domain depends on the value of $N_{\text{max}}$.

\textit{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.

\textit{Calculate the size of the input domain}
Equivalence Class 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, as shown in (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*. 
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 allows 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. $E_1$, $E_2$, $E_3$, and $U_1$, $U_2$).
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 $\text{age}$ as input. Let us suppose that the only legal values of $\text{age}$ 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 $R_1$ and those in the range $[62..120]$ according to requirement $R_2$.

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.)

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 $R_1$. Similarly, any test selected from the region [62..120] will reveal any fault with respect to $R_2$. A similar expectation applies to the two regions containing the unexpected inputs.

**Question:** Is this assumption correct?

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 $f$. Using the partitioning method described in the previous example, we obtain the following equivalence classes.
Example 2 (contd.)

<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, not empty</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, not empty</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>

Note that the number of equivalence classes without any knowledge of the program code is 2, whereas the number of equivalence classes on slide 25 is 6.

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:

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

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>Classes</td>
</tr>
<tr>
<td>One class with values inside the range and two with values outside the range.</td>
<td>speed ( \in [60..90] )</td>
</tr>
<tr>
<td>area: float</td>
<td>area ( \geq 0.0 )</td>
</tr>
<tr>
<td>age: int</td>
<td>{{-1}, {56}, {132}}</td>
</tr>
</tbody>
</table>

Equivalence Classes for Variables: Strings

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td></td>
</tr>
<tr>
<td>At least one containing all legal strings and one all illegal strings based on any constraints.</td>
<td>firstname: string</td>
</tr>
</tbody>
</table>
## Equivalence Classes for Variables: Enumeration

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
<th>Constraints</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each value in a separate class</td>
<td>autocolor:{red, blue, green}</td>
<td></td>
<td>{{red}, {blue}, {green}}</td>
</tr>
<tr>
<td>up:boolean</td>
<td></td>
<td>{{true}, {false}}</td>
<td></td>
</tr>
</tbody>
</table>

## Equivalence Classes for Variables: Arrays

<table>
<thead>
<tr>
<th>Eq. Classes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Constraints</td>
</tr>
<tr>
<td>One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.</td>
<td>int [ ] aName = new int[3];</td>
</tr>
</tbody>
</table>
Equivalence Classes for Variables: Compound Data Type

Objects and 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 example illustrates the derivation of equivalence classes for an input variable that has a compound type.

Equivalence Classes for Variables: Compound Data Type: Example

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

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

Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually. Many classes so created might be infeasible. Nevertheless, equivalence classes so created offer an increased variety of tests as is illustrated in the next section.
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.)**

$$
\begin{align*}
E1: & \quad x < 3 \\
E2: & \quad 3 \leq x \leq 7 \\
E3: & \quad x > 7 \\
E4: & \quad y < 5 \\
E5: & \quad 5 \leq y \leq 9 \\
E6: & \quad y > 9
\end{align*}
$$

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

E1: \(x<3, y<5\)  \hspace{1cm} E2: \(x<3, 5 \leq y \leq 9\)  \hspace{1cm} E3: \(x<3, y>9\)

E4: \(3 \leq x \leq 7, y<5\)  \hspace{1cm} E5: \(3 \leq x \leq 7, 5 \leq y \leq 9\)  \hspace{1cm} E6: \(3 \leq x \leq 7, y>9\)

E7: \(x>7, y<5\)  \hspace{1cm} E8: \(x>7, 5 \leq y \leq 9\)  \hspace{1cm} E9: \(x>7, y>9\)

Partitioning Example (contd.)

6 equivalence classes

9 equivalence classes
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*.

2. **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 behave in the *same way* are grouped together.

Note that “same way” needs to be defined by the tester.
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.

---

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.
GUI Design and Equivalence Classes

While designing equivalence classes for programs that obtain input exclusively from the user, 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 (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 figures 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 $f_1$ when $x \leq 0$ and function $f_2$ otherwise. 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 $x > 0$.

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*. 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: Step 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: Step 2. Identify Boundaries

Equivalence classes and boundaries for \textit{findPrice}. Boundaries are indicated with an x.

\begin{center}
\begin{tabular}{cccc}
98 & 100 & 998 & 1000 \\
\hline
* & x & * & x & * \\
\hline
E1 & & 99 & & 999 & E3 \\
E2 & & & & \\
0 & 2 & 99 & 101 \\
\hline
\end{tabular}
\end{center}

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. Consider the following test set:

\begin{verbatim}
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)
}
\end{verbatim}

Question: does this example use unidimensional or multidimensional partitioning?