Domain Testing Models

- Domain analysis is a straightforward and effective way to select test values
- There are several domain testing models:
  - Meyer’s equivalence class
  - ...
  - Used in book: based on White and Cohen’s paper in 1978
Domain Analysis

- A domain is the set of all inputs accepted by the IUT.
- A subdomain is a partition of the domain defined by boundary conditions:
  - Boolean expressions on the IUT input variables
  - Message parameters and instance variables
- Fault model for domain testing?
  - IUT has incorrectly implemented a boundary.
- Example:
  - Collection class with a maximum number of 4096 items. The IUT allows 4095 items but reject 4096.
Typical domain faults

<table>
<thead>
<tr>
<th>Domain Fault</th>
<th>Required</th>
<th>Incorrect Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shifted Boundary</td>
<td>( y &lt; 0 )</td>
<td>( y &lt; 1 )</td>
</tr>
<tr>
<td>Tilted Boundary</td>
<td>( y \leq 10 - x )</td>
<td>( y \leq 10 + x )</td>
</tr>
<tr>
<td>Missing Boundary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extra Boundary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 10.18  Domain fault model.
Domain Analysis (cont)

- A domain test suite is developed by domain analysis
  - Identify constraints for all input variables
  - Select test values for each variable in each boundary
  - Select test values for variables not given in the boundary
  - Determine expected results for these input
- The results are represented in the domain matrix
- Example:
  - Assume that function `void aFunction(int x, float y, Stack aStack)` has the following constraints:
    ```c
    assert ( (y >= 1.0)     && // condition 1
             (x <= 10)      && // condition 2
             (y <= 10.0)    && // condition 3
             (x > 0)        && // condition 4
             (y <= 14.0 –x) && // condition 5
             (! aStack.isFull) // condition 6
    ```
Condition 1: 
\((x > 0)\)

Condition 2: 
\((x \leq 10)\)

Condition 3: 
\((y \geq 10)\)

Condition 4: 
\((y \leq 10)\)

Condition 5: 
\((y \leq (14 - x))\)
On, Off, In, Out

- All domain testing selection criteria use points on and off points

- With respect to a particular boundary
  - On points lie on the boundary.
  - Off points lie off the boundary.

- With respect to all boundaries
  - In points satisfy all boundary conditions & are not on a boundary.
    - Normally used as typical values
  - Out points satisfy no boundary conditions & are not on a boundary.
    - Not that useful
Open and closed boundaries

- **Open Boundary**
  - Uses strict inequality (e.g., \( x > 0 \))
  - **On** point:
    - \( x=0 \), makes boundary condition false.
  - **Off** point
    - Makes boundary condition false
    - must be inside the domain: \( x=1 \)

- **Closed Boundary**
  - Uses strict equality (e.g. \( y \leq 10.0 \))
  - **On** point:
    - \( y=10.0 \), makes boundary condition true
  - **Off** point:
    - Make the condition false: \( y=10.000001 \)
What about OO systems?

- For primitive data types, applying domain analysis is straightforward
- Can we apply it to classes?
- No, when domain boundaries are defined with complex classes:
  - We cannot focus on a single variable of such objects
  - Must consider all instance variables
- Solution: For OO systems, both external inputs and the state of the object must be considered
An example

- Consider the following class

```java
class Account {
    AccountNumber number;
    Money balance;
    Date lastUpdate;
    ...
}
```

- A primitive view of the state space would yield too many states…
Trillions of states

**FIGURE 7.16** Primitive view of a state space.
State invariants

- State: A set of variable value combinations that share some property of interest
- A valid state can be expressed with a state invariant,
  - Boolean expression that can be checked
- State invariant and class invariant?
  - Class invariant is weaker
- Domain testing model to objects:
  - Treat each state invariant as a domain boundary
OO domain definitions

- An abstract state **on** point is a state such that the smallest possible change in any variable will produce a state change.
- An abstract state **off** point is a valid state that is not the focus state and differs from the focus state by the smallest possible change.
- An abstract state **in** point is neither an **on** or an **off** point.
Example: Stack class

- Abstract states of Stack:
  - *empty* (Stack.size() == 0)
  - *loaded* (0 < Stack.size() < MAXSTACK)
  - *full* (Stack.size() == MAXSTACK)

Assuming MAXSTACK = 32767

<table>
<thead>
<tr>
<th>State</th>
<th>Possible Transitions</th>
<th>In Point</th>
<th>On Point</th>
<th>Off Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>loaded</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>loaded</td>
<td>empty, full</td>
<td>1&lt;x&lt;32766</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>1&lt;x&lt;32766</td>
<td>32766</td>
<td>32767</td>
</tr>
<tr>
<td>full</td>
<td>loaded</td>
<td>32767</td>
<td>32767</td>
<td>32766</td>
</tr>
</tbody>
</table>
The One by One Selection Criteria

- 1 x 1 (“one by one”) domain testing strategy calls for one on point and one off point for each domain boundary.

- Using the abstract state model, 1 x 1 is applicable to the full range of data types used in object oriented programming.
The 1x1 Selection Criteria - 1

- For relational conditions, e.g. $x \leq 10$
  - One **on** point and one **off** point
- For strict equality conditions, e.g. $x = 10$
  - One **on** point and two **off** points
- For non-scalar types: one **on** point and one **off** point
- For abstract state invariants: one **on** point and at least one **off** point
- For nonlinear boundaries: one **on** point and one **off** point
- Don’t repeat identical tests for adjacent subdomains
An example

```c
void aFunction(int x, float y, Stack aStack){
    assert( (y >= 1.0) &&
            (x <= 10) &&
            (y <= 10.0) &&
            (x > 0) &&
            (y <= 14.0 - x) &&
            (!aStack.isFull()));

    ......
}
```
Domain Matrix Design

- Define **on** and **off** points for each boundary condition
- Add expected results and **in** points for other values
  - Each test case only has one **off** or **on** point
  - Select **in** points for all other values in the test case
  - Avoid to repeat **in** points. Why?

**Result**
- Minimal test cases
- Input variables to exercise boundary conditions
- For any type of variable types
  - Including abstract complex types (objects)
Error in the figure: The value of x for test case 9 must be 7, so that y is an off point. The in point for y test cases 3, 4 and 9 is also wrong (all invalidate x <=14-x).
What does this approach achieve?

- This is a systematic sampling approach to test case design
- We can’t afford to run all tests, so we divide the population of tests into subpopulations and test one or a few representatives of each subgroup
- This keeps the number of tests manageable
What does this approach achieve?

- Using *boundary values* for the tests offers a few benefits:
  - They will expose any errors that affect an entire equivalence class.
  - They will expose errors that mis-specify a boundary.
    - These can be coding errors (off-by-one errors such as saying “less than” instead of “less than or equal”) or typing mistakes (such as entering 57 instead of 75 as the constant that defines the boundary).
    - Mis-specification can also result from ambiguity or confusion about the decision rule that defines the boundary.
  - Non-boundary values are less likely to expose these errors.