Embedded Functions in Combinatorial Test Designs

George B. Sherwood
This talk is about:

• Conforming to constraints in combinatorial test designs

• A feature to improve usability and adoption of combinatorial testing by practitioners

• Embedded functions using a general-purpose programming language
  – Combination functions to define constraints for test case generation
  – Substitution functions to evaluate expected equivalence class(es) of test cases

• Ongoing project to assess usability and performance
Test model terms

• There are $k$ test factors, e.g. configurations & inputs
• A test case has 1 value for each factor (i.e. a $k$-tuple)
• A strength-$t$ design ($t \leq k$) includes all required $t$-tuples of test factor values
• A partition includes the allowed combinations ($t$-tuples) for 1 test case generation instance
• An equivalence class includes combinations for 1 class of expected results
• A partition’s test cases can exercise 1 or more equivalence classes
Why are constraints needed?

- To cover all required configurations
- To cover all required input combinations
- To cover required tuples for different classes of results

All 3 situations can cause *masking*:
A required $t'$-tuple (with $t' \leq t$) is missing from the test cases for a class of results
Constraints are needed to cover all required tuples
Functionally dependent test factor values

• Constraints can be described using functionally dependent test factor values

• Functional dependence:
  1 or more values of a dependent factor are identified by those of other, determinant factors
  Determinant factors’ values → dependent factor values

• Example: The last day of any month is identified by the month and its year
  Month, Year → Last day values
  \( l = \) number of determinant factors (\( l = 2 \) in this example)

• Use Direct Product Block (DPB) notation with or without embedded combination functions
Direct Product Block (DPB) Notation

Fixed values form

Calendar Example without last_day function
Month
Day
Year
#ok All good dates
jan feb mar apr may jun jul aug sep oct nov dec
1 10
2015 2016 2017
+ long month last day
jan mar may jul aug oct dec
31
2015 2016 2017
+ short month last day
apr jun sep nov
30
2015 2016 2017
+ feb last day
feb
28
2015 2017
+ leap day
feb
29
2016

• Valid calendar dates example with boundary checking
• Factor values are on separate lines
• All combinations in a block are allowed
• Partition of multiple blocks includes union of their allowed combinations
Direct Product Block (DPB) Notation

Fixed values form

Calendar Example without last_day function
Month
Day
Year
#ok All good dates
jan feb mar apr may jun jul aug sep oct nov dec
1 10
2015 2016 2017
+ long month last day
jan mar may jul aug oct dec
31
2015 2016 2017
+ short month last day
apr jun sep nov
30
2015 2016 2017
+ feb last day
feb
28
2015 2017
+ leap day
feb
29
2016

Functionally dependent form

Calendar Example with last_day function
$month
Day
$year
#ok All good dates
jan feb mar apr may jun jul aug sep oct nov dec
1 10 last_day($month,$year)
2015 2016 2017

- Month, Year → Last day values
- Factors renamed as variables for function arguments:
  $month $year
- Day values:
  1 10 last_day($month,$year)
- 5 blocks now represented by only 1 block
Combination functions

- `last_day($month,$year)` is a combination function
- Combination functions return dependent values for all allowed combinations of determinant factor values
- Generator uses these fixed values to construct test cases
- `last_day($month,$year)` needs to return the last day for any month in the years 2015 2016 2017
- PHP built-in function `cal_days_in_month` is reused:

```php
<?php
function last_day($month,$year) {
    $mo_num=array('jan'=>1,'feb'=>2,'mar'=>3,'apr'=>4,'may'=>5,'jun'=>6,'jul'=>7,'aug'=>8,'sep'=>9,'oct'=>10,'nov'=>11,'dec'=>12);
    return(cal_days_in_month(CAL_GREGORIAN,$mo_num[$month],(int)$year));
}
?>
```
Why PHP?

- Other languages can support embedded functions
- Prevalence: millions of programmers
  - Free, open source
  - Easy to learn
- Support for user-defined (embedded) functions
- Hundreds of built-in functions, for reuse as needed
- Good performance without explicit compilation
## Constraint simplification

### Instant Shopping

Your cart contains:

<table>
<thead>
<tr>
<th>Delete</th>
<th>Item Number</th>
<th>Item Description</th>
<th>Quantity</th>
<th>Price</th>
<th>Item Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>itemA</td>
<td>descriptionA</td>
<td>1</td>
<td>14.95</td>
<td>14.95</td>
</tr>
<tr>
<td></td>
<td>itemB</td>
<td>descriptionB</td>
<td>2</td>
<td>9.95</td>
<td>19.90</td>
</tr>
<tr>
<td></td>
<td>itemC</td>
<td>descriptionC</td>
<td>1</td>
<td>5.95</td>
<td>5.95</td>
</tr>
</tbody>
</table>

Subtotal: $40.80

### Constraints:
- Items in different positions must be different
- Factor values may be NULL (unused)
- Equivalence classes are target UML leaf states to avoid masking

### Simplification for nonemptyCart to nonemptyCart transition:
- Without combination functions: 33 blocks
- With 7 combination functions: 3 blocks
- Average function length: 7 lines
Shopping cart state diagram

- **entry / addItem();**
- **exit / for(i=0; i<n; i++) {**
  - **delChk[i]=0;**
  - **newQ[i]=qty[i];**
- **}**

- **[n>0]**
- **[else]**

- **emptyCart**
  - **entry / displayCartStart();**
  - **displayButton(SHOP);**
  - **displayCartEnd();**

- **nonemptyCart**
  - **entry / displayNonemptyCart();**

- **CHECKOUT**
- **UPDATE / updateItems();**
- **SHOP**

- **CHECK(i)[0<=i&&i<n] / delChk(i)=1-delChk(i);**
- **QTY(i,q)[0<=i&&i<n] / newQ[i]=q;**
## Shopping cart test factors

### Test factors:
- **Program variables**
  - Indexed for cart position
  - NULL values as needed
- **Current state**
  - nonemptyCart
- **Event (trigger)**
  - CHECK QTY UPDATE

### Combination functions
- Composite functions as needed
- Same function for each indexed factor in each block

### Table: Test Factors and Values

<table>
<thead>
<tr>
<th>Test Factor</th>
<th>Test Factor Values</th>
<th>Combination Functions</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>$newItem</td>
<td>NULL</td>
<td></td>
<td>Item to place in cart</td>
</tr>
<tr>
<td>$n</td>
<td>1 2 3</td>
<td></td>
<td>Number of items in cart</td>
</tr>
<tr>
<td>$delChk[0]</td>
<td>0 1</td>
<td>f_delChk</td>
<td>Delete box checked in cart position 0</td>
</tr>
<tr>
<td>$item[0]</td>
<td>itemA itemB itemC</td>
<td>f_item</td>
<td>Item in cart position 0</td>
</tr>
<tr>
<td>$qty[0]</td>
<td>1 2 10</td>
<td>f_qty</td>
<td>Quantity of item in cart position 0</td>
</tr>
<tr>
<td>$newQ[0]</td>
<td>0 1 2 10</td>
<td>f_newQ</td>
<td>New quantity shown in cart position 0</td>
</tr>
<tr>
<td>$delChk[1]</td>
<td>0 1 NULL</td>
<td>f_delChk</td>
<td>Delete box checked in cart position 1</td>
</tr>
<tr>
<td>$item[1]</td>
<td>itemA itemB itemC</td>
<td>f_item</td>
<td>Item in cart position 1</td>
</tr>
<tr>
<td>$qty[1]</td>
<td>1 2 10 NULL</td>
<td>f_qty</td>
<td>Quantity of item in cart position 1</td>
</tr>
<tr>
<td>$newQ[1]</td>
<td>0 1 2 10 NULL</td>
<td>f_newQ</td>
<td>New quantity shown in cart position 1</td>
</tr>
<tr>
<td>$delChk[2]</td>
<td>0 1 NULL</td>
<td>f_delChk</td>
<td>Delete box checked in cart position 2</td>
</tr>
<tr>
<td>$item[2]</td>
<td>itemA itemB itemC</td>
<td>f_item</td>
<td>Item in cart position 2</td>
</tr>
<tr>
<td>$qty[2]</td>
<td>1 2 10 NULL</td>
<td>f_qty</td>
<td>Quantity of item in cart position 2</td>
</tr>
<tr>
<td>$newQ[2]</td>
<td>0 1 2 10 NULL</td>
<td>f_newQ</td>
<td>New quantity shown in cart position 2</td>
</tr>
<tr>
<td>$i</td>
<td>0 1 2 NULL</td>
<td>f_i</td>
<td>Cart position for event</td>
</tr>
<tr>
<td>$q</td>
<td>0 1 2 10 NULL</td>
<td></td>
<td>Quantity for event</td>
</tr>
<tr>
<td>state</td>
<td>nonemptyCart</td>
<td></td>
<td>Source state</td>
</tr>
<tr>
<td>event</td>
<td>CHECK($i) QTY($i,$q) UPDATE</td>
<td>f_event_CHECK f_event_QTY</td>
<td>Trigger to target state</td>
</tr>
</tbody>
</table>
Substitution functions

• A substitution function returns a value for each test case after test case generation
• A substitution function value can be determined by its test case factor values
• Substitution functions for equivalence classes can identify the expected class for each test case
  – To evaluate expected equivalence classes automatically
  – To enable equivalence class coverage assessment
  – To check pre-generation equivalence class analysis
• Why equivalence classes?
Body mass index report requirements

R1. Input data for patient database table:
   - Age in years
   - Weight in pounds
   - Height in inches
   - Sex (female, male)
   - Intake in kilocalories per day

R2. Compute & store body mass index:
   \[ BMI = \frac{703 \times \text{Weight}}{\text{Height}^2} \]

R3. Age ≥ 65: Generate Medicare report

R4. Age < 20 Generate Child report:
   - Girl, percentile from female BMI-age table
   - Boy, percentile from male BMI-age table

R5. Age ≥ 20 Generate Adult report:
   - Underweight, BMI < 18.5
   - Normal, 18.5 \leq BMI < 25.0
   - Overweight, 25.0 \leq BMI < 30.0
   - Obese, 30.0 \leq BMI
Why equivalence classes?

• Equivalence classes group test factor combinations by similar expected results

• Classes help insure test design coverage
  Example: The Medicare, Child and Adult reports each have multiple, valid equivalence classes

<table>
<thead>
<tr>
<th>Report</th>
<th>Valid equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medicare</td>
<td>no</td>
</tr>
<tr>
<td>Child</td>
<td>no</td>
</tr>
<tr>
<td>Adult</td>
<td>no</td>
</tr>
</tbody>
</table>

• Equivalence classes are functionally dependent
  Input, configuration values → result → equivalence class

• Report classes can be expressed as 3 functions
Equivalence class considerations

• Partitions are associated with 1 or more equivalence classes, based on test models and goals

• Required equivalence classes must be reached:
  – Either $t \geq l$ in a multi-class partition, with $l$-tuples for all required classes
  – Or the partition is constrained to 1 class, with $l$-tuples for that class

Examples:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Class Type</th>
<th>$l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Medicare classes</td>
<td>1</td>
</tr>
<tr>
<td>Age, Sex</td>
<td>Child classes</td>
<td>2</td>
</tr>
<tr>
<td>Age, Weight, Height</td>
<td>Adult classes</td>
<td>3</td>
</tr>
</tbody>
</table>

• Class boundaries are frequently locations of programming errors
Equivalence class associations

• Each class may be associated with other factors according to its *nondeterminant strength* $s$
  – Either $s = t - l + 1$ in a multi-class partition
  – Or $s = t + 1$ in a single-class partition

Examples with strength $t = 3$:

<table>
<thead>
<tr>
<th>Factor(s)</th>
<th>$l$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age → Medicare classes</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Age, Sex → Child classes</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Age, Weight, Height → Adult classes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• To test all classes with their nondeterminant factors:
  – Either use $s \geq 2$ ($t \geq 4$ in this example) in a multi-class partition
  – Or use a lower strength in single-class partitions
## Equivalence class substitution functions

Substitution functions help assess equivalence class coverage.
Embedded functions

• A feature to improve usability and adoption of combinatorial testing by practitioners

• Uses a general-purpose programming language for
  – Combination functions to define constraints for test case generation
  – Substitution functions to evaluate expected equivalence class(es) of test cases

• Ongoing project to assess usability and performance