Embedded Functions in Combinatorial Test Designs

George B. Sherwood

This talk is about:

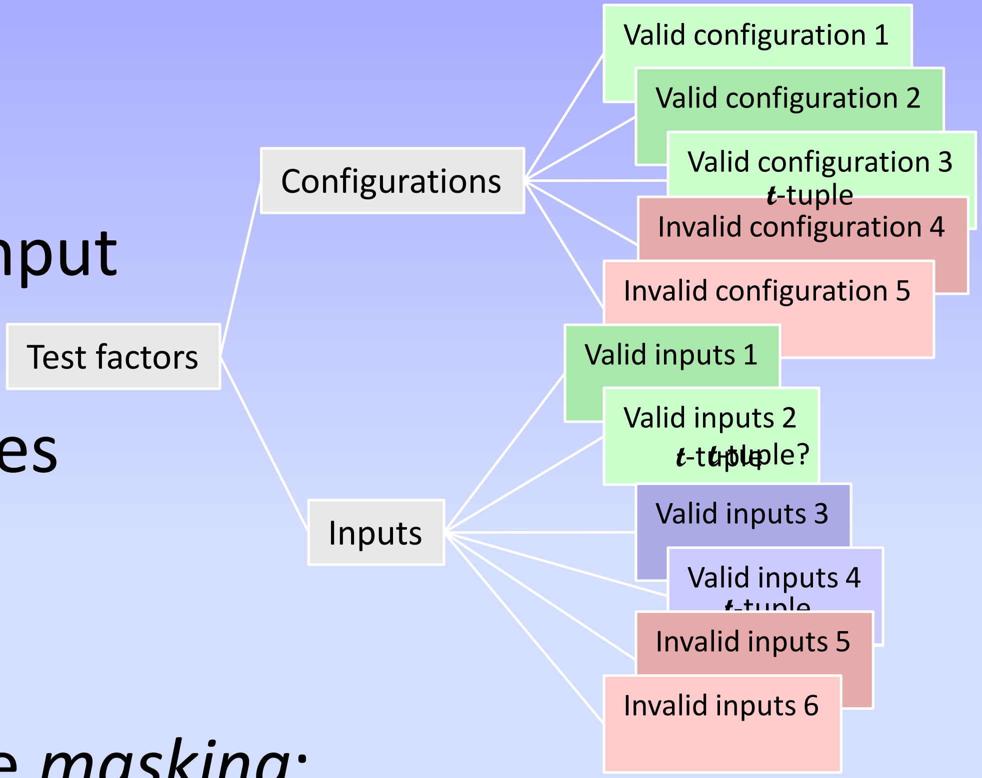
- Conforming to constraints in combinatorial test designs
- A feature to improve usability and adoption of combinatorial testing by practitioners
- Embedded functions using a general-purpose programming language
  - Combination functions to define constraints for test case generation
  - Substitution functions to evaluate expected equivalence class(es) of test cases
- Ongoing project to assess usability and performance

Test model terms

- There are $\boldsymbol{k}$ test factors, e.g. configurations & inputs
- A test case has 1 value for each factor (i.e. a $\boldsymbol{k}$-tuple)
- A strength-$\boldsymbol{t}$ design ($\boldsymbol{t} \le \boldsymbol{k}$) includes all required $\boldsymbol{t}$-tuples of test factor values
- A *partition* includes the allowed combinations ($\boldsymbol{t}$-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



Test factors

Configurations

Valid configuration 1

Valid configuration 2

Valid configuration 3

t-tuple

Invalid configuration 4

Invalid configuration 5

Inputs

Valid inputs 1

Valid inputs 2

t-tuple

t-tuple?

Valid inputs 3

Valid inputs 4

t-tuple

Invalid inputs 5

Invalid inputs 6

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 ($\ell = 2$ 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

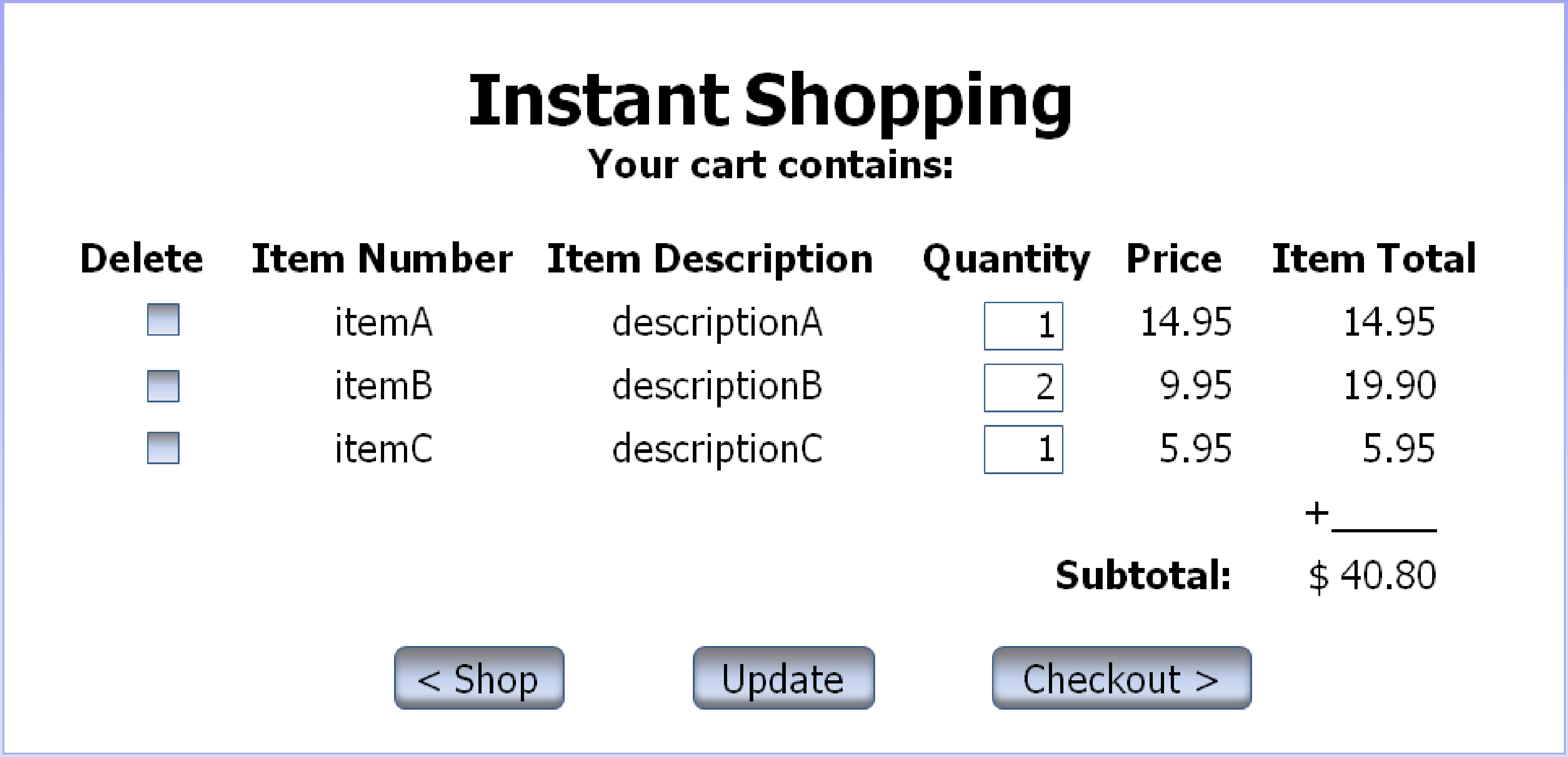
- last_day($month,$year) is a combination function
- Combination functions return dependent values for all allowed combinations of determinant factor values
- Generator uses these fixed values to construct test cases
- last_day($month,$year) needs to return the last day for any month in the years 2015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

```
<?php
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:

Delete	Item Number	Item Description	Quantity	Price	Item Total
☐	itemA	descriptionA	1	14.95	14.95
☐	itemB	descriptionB	2	9.95	19.90
☐	itemC	descriptionC	1	5.95	5.95
					+
				Subtotal:	$ 40.80

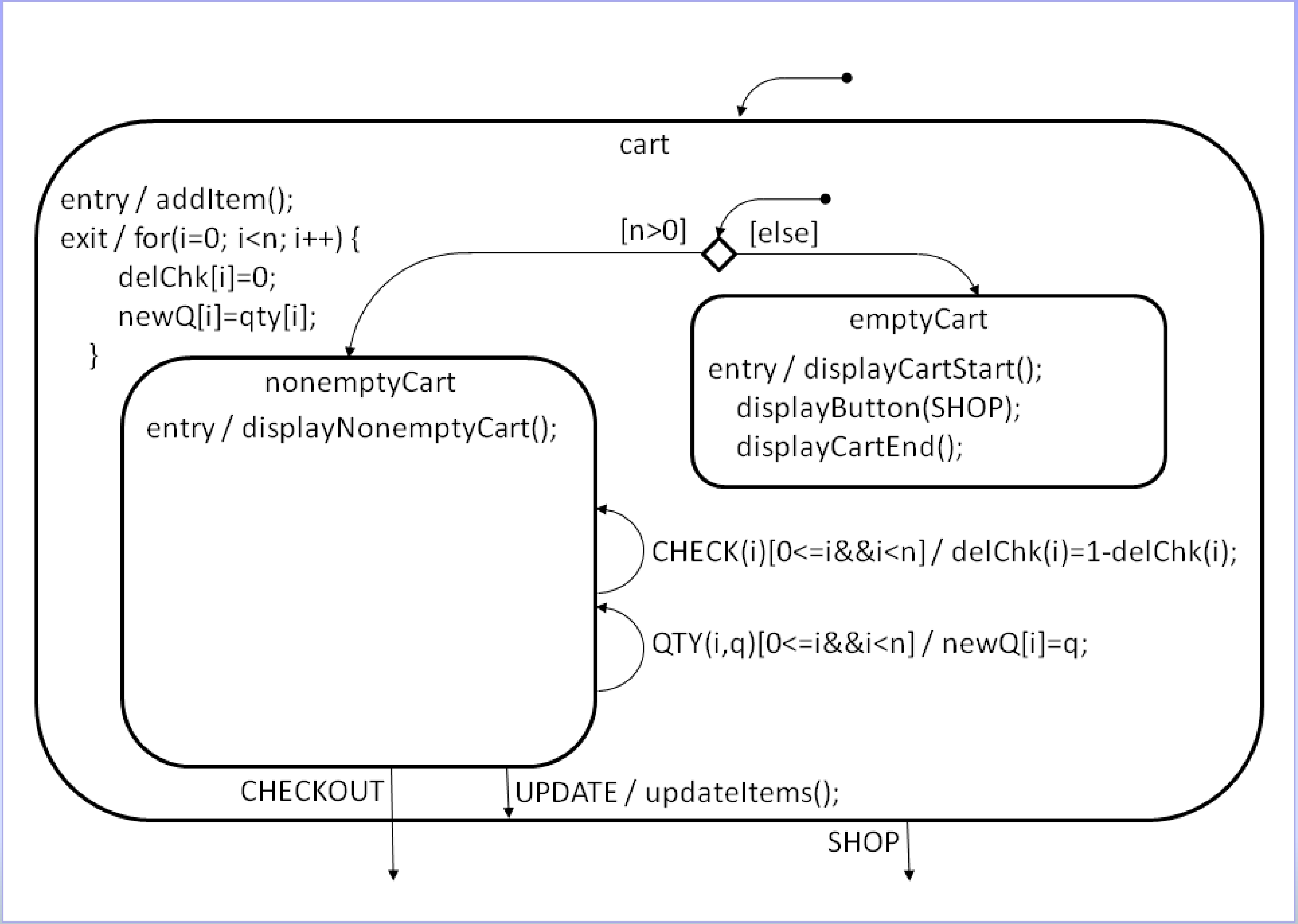
Constraints:

- 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



cart

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();

CHECK(i)[0<=i&&i<n] / delChk(i)=1-delChk(i);

QTY(i,q)[0<=i&&i<n] / newQ[i]=q;

CHECKOUT

UPDATE / updateItems();

SHOP

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

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

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:

$\text{BMI} = 703 \text{ x } \text{Weight} / \text{Height}^2$

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 ≤ BMI < 25.0
Overweight, 25.0 ≤ BMI < 30.0
Obese, 30.0 ≤ 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

Report	**Valid equivalence classes**				
Medicare	no	yes			
Child	no	girl	boy		
Adult	no	underweight	normal	overweight	obese

- 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 $t \ge \ell$ 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:

		ℓ
Age	→ Medicare classes	1
Age, Sex	→ Child classes	2
Age, Weight, Height	→ Adult classes	3

- 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 $s = t - \ell + 1$ in a multi-class partition
  - Or $s = t + 1$ in a single-class partition

Examples with strength $t = 3$:

		ℓ	s
Age	→ Medicare classes	1	3
Age, Sex	→ Child classes	2	2
Age, Weight, Height	→ Adult classes (Nondeterminant: Sex, Intake)	3	1

- 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

Test cases with equivalence classes evaluated

Input factors						Equivalence class factors		
Test Case	***Age***	***Weight***	***Height***	***Sex***	***Intake***	***Medicare***	***Child***	***Adult***
1	19	131	64	female	2000	no	girl	no
2	19	131	64	male	3000	no	boy	no
3	19	131	71	male	2000	no	boy	no
4	19	180	64	female	3000	no	girl	no
5	19	180	71	female	2000	no	girl	no
6	19	180	71	male	3000	no	boy	no
7	42	131	64	female	2000	no	no	normal
8	42	131	64	male	3000	no	no	normal
9	42	131	71	female	3000	no	no	underweight
10	42	180	64	male	2000	no	no	obese
11	42	180	71	female	2000	no	no	overweight
12	42	180	71	male	3000	no	no	overweight
13	67	131	64	female	2000	yes	no	normal
14	67	131	64	male	3000	yes	no	normal
15	67	131	71	female	3000	yes	no	underweight
16	67	180	64	male	2000	yes	no	obese
17	67	180	71	female	2000	yes	no	overweight
18	67	180	71	male	3000	yes	no	overweight

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