Embedded Functions in Combinatorial Test Designs

George B. Sherwood
Testcover.com, LLC
Colts Neck, NJ USA
sherwood@testcover.com

***Abstract*—A feature for conforming to system constraints during combinatorial test case generation is proposed. The user interface and requirements for the feature are outlined. For constraint conformance, allowed test factor values are given by functions embedded in the test case generator. The functions are defined in a general-purpose programming language widely used among software engineers. The language, PHP, is chosen for its flexibility and prevalence. Examples of functions conforming to constraints are given. A second type of embedded function is proposed to automate the identification of expected equivalence class(es) for each test case. Equivalence class functions return the classes according to test case values. Examples of equivalence class functions are given also. Implementation of the feature is ongoing; goals include assessment of the feature's feasibility, usability and performance.**

Keywords—combinatorial testing; constraints; coverage analysis; embedded function; equivalence class; equivalence partitioning; functional dependence; interaction testing; PHP; test case generation; test design

I. INTRODUCTION

Combinatorial testing is a powerful technique to find small sets of test cases for complex systems. Such a system has a number k of test factors (e.g. test configuration choices and input parameters). Typically, testing all combinations of test factor values is infeasible because there are too many test cases to execute and analyze. However, by using the combinations of smaller sets of test factor values (t-tuples, with $t < k$), the number of test cases can be significantly reduced.

The need to accommodate system constraints in combinatorial testing was recognized early in its development [1,2], and considerable research and tool development progress has ensued. But today, decades later, generating test cases consistent with system constraints is an ongoing challenge to the broad adoption of combinatorial testing. The challenge presents itself in two ways.

The first challenge is very apparent: Test cases must be "valid." Configurations that are impossible to set up are not to be considered. A condition that always leads to an abnormal result, like an error report, should be tested. But it may not be a good candidate for combinatorial testing if it leads to redundant test cases (e.g. if one factor value always gives the error). More importantly, if a t-tuple is in only one test case, for an abnormal result, that t-tuple will not be used to test the normal operation of the system. This is a form of *masking* [1,3,4], which diminishes the expected benefit of covering all t-tuples. Masking occurs when a required t'-tuple (with $t' \leq t$) is missing from the test cases for a class of results.

The second challenge is more subtle: Not all valid test cases are equivalent. An *equivalence class* includes the combinations of test factor values for one class of expected results. A *partition* includes the allowed combinations of test factor values for a test case generation instance. When test models do not adequately partition the equivalence classes of valid results, masking can still occur. It is possible that a t-tuple which leads to an expected result in one equivalence class will miss finding a fault in another class if it is applied in only one test case. For example, a strength-3 design in a single partition can miss pairs that are covered by a strength-2 design with adequate partitioning [5]. The alternative approach, to increase the strength of the design and keep a single partition, quickly becomes impractical. The exponential increase in test cases multiplies the work beyond the budgets and schedules of the responsible practitioners.

The research community has had some success addressing these challenges using a variety of approaches, including model checkers, Boolean satisfiability solvers, satisfiability modulo theories solvers, formal logic and other new algorithms [6-13]. However, these techniques are not broadly known or used in the software engineering community.

This paper proposes that to achieve the benefits of combinatorial testing, broader adoption is needed. And for broader adoption, combinatorial test tools need to implement constraints and to model equivalence classes in a manner more accessible to the software engineering community. The paper introduces a new feature to do so, embedded functions.

In the course of a development project, the system's requirements and implementation need to drive test planning and execution for a successful outcome. This project information is the basis for specifying test constraints and equivalence classes. If the constraints can be expressed in a flexible language familiar to software engineers, advantages in usability and efficiency may result. The idea behind embedded functions is to use a programming language prevalent in the software engineering community to define the constraints as functions. The test case generator then uses these functions to conform to the constraints.

The project described here is ongoing. The paper outlines feature requirements, examples of using embedded functions and anticipated benefits. The goal of the project is to assess the feasibility of the embedded functions feature, and its usability and performance characteristics. Conclusions about this assessment are beyond the current scope.

The remainder of the paper is organized into 4 sections. Section II describes the embedded functions feature in more

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
4th International Workshop on Combinatorial Testing (IWCT 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

detail. Section III contains examples of *combination functions*, whose evaluation define the constraints for test case generation. Section IV illustrates the use of *substitution functions* to evaluate the expected equivalence class(es) for each test case [5]. Section V provides a brief discussion of goals.

II. Feature Description

The general idea of embedded functions is to model system constraints and equivalence classes using a well known, frequently used programming language. PHP [14] is the language selected for this project. PHP is broadly used as a server-side scripting language for internet applications. It supports user-defined functions and contains a comprehensive set of built-in functions.

A. User Interface

The embedded functions feature enhances Direct Product Block (DPB) notation to accept factor values returned from PHP functions. (DPB notation is introduced in Section 6.1.3 of [15].) The test case generation request is entered in DPB notation, in a text box on the generator form. User-defined PHP functions are entered via two other text boxes, one for combination functions and one for substitution functions.

In DPB notation the allowed values for each factor appear on a separate line in a block. All combinations of values in the block are allowed. Partitions have one or more blocks. A partition's allowed combinations include all of its blocks' combinations. The blocks are defined so that they do not include any disallowed combinations. The first panel of Table I gives an example of DPB notation to specify valid dates according to calendar rules.

In the enhanced DPB notation, functions are listed as values for dependent factors. In the second panel of Table I, the Day factor has 2 fixed values, 1 and 10, and the function last_day($month,$year). The function name must conform to PHP rules [14] and start with a letter or underscore (_). The arguments to the function are the determinant factor names $month and $year. Generally factor names are optional in DPB notation. However names for determinant factors are required to specify function arguments. Determinant factor names are PHP variables[1], so they are represented by a dollar sign ($) followed by the name of the variable. Function arguments can be determinant factor names or fixed values.

The first two panels of Table I show that the last_day function allows the number of blocks to be reduced from 5 to 1. It is simple to define this function to return the last day for the specified years. Alternatively, the third panel of Table I shows how the built-in function cal_days_in_month [14] can support a broader range of years.

TABLE I. Combination Function Example

Calendar Partition with Fixed Values

```
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
```

Calendar Partition with Combination Function last_day($month,$year)

```
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
```

PHP Function last_day($month,$year)

```
<?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));
}
?>
```

B. Combination Functions

Combination functions define the constraints for test case generation. They represent test factor values in functionally dependent form [5]. The embedded functions feature automates the procedure to convert the request to fixed values form. Then a proprietary algorithm is used to generate test cases consistent with the constraints.

The last_day function is a combination function to be evaluated before test case generation. Each combination function is called for all combinations of its arguments' values. The function returns one allowed value, or a list of them, for generating test cases. Combination functions should return values only for allowed combinations of arguments. Disallowed combinations include impossible configurations, inputs that do not result in the desired equivalence class(es), etc. This use of allowed combinations can avoid masking and can facilitate appropriate equivalence class partitioning in well defined systems under test.

When a function cannot be evaluated before test case generation, it is treated as a fixed value during generation. This

[1] All variables are for the test model; they do not necessarily correspond to the implementation of the system under test.

will happen for substitution functions, which are evaluated after generation, for each test case.

C. Substitution Functions

Currently some tools allow substitutions following test case generation. Typically these include replacing values in a set of test cases with random or unique values. In these situations, the different values are considered unimportant for the combinatorial coverage but are required for the test design. For example, user IDs need to be unique.

The substitution functions proposed here extend this idea to enable a variety of processing tasks after test case generation. Each substitution function is expected to return a single value for any allowed combination of its arguments. This value is substituted into the associated test case.

Substitution functions for *equivalence class factors* [5] can be used to verify coverage of classes of results. Equivalence classes, defined by requirements, are used to organize expected results into classes which are equivalent for test purposes. For example, the production of a Medicare report defines an equivalence class. When $Age is greater than or equal to 65 the report is produced; when $Age is less than 65 it is not. An *equivalence class function* can return the class of the expected result based on the value(s) of determinant test factor(s). In this example the function Medicare_report($Age) would return an allowed value for this partition, 'yes' or 'no' accordingly.

The first panel of Table II shows 2 test cases adapted from Table XII of [5]. The Medicare equivalence class factor has a single value, Medicare_report($Age), a function which is not evaluated before test cases are generated[2]. Evaluation after generation provides the class of the expected result. The PHP function Medicare_report($Age) is defined in the second panel of Table II. It returns the allowed equivalence class values for this partition. It does not return a value for $Age -5, which does not belong in this partition. When a substitution function does not return any value, the function with arguments, Medicare_report(-5), is listed in the test case instead. The intention is to identify test design issues, such as disallowed combinations, or errors in function definitions.

Reference [5] shows how coverage of the equivalence classes can be assessed, even before test cases are generated. Evaluation of equivalence class functions after test case generation enables verification of the coverage analysis. Thus coverage omissions and masking can be avoided when the classes of expected results are explicit. And the automatic evaluation enhances convenience and accuracy.

[2] Because the equivalence class factor has only one value during test case generation, it does not increase the number of test cases.

TABLE II. SUBSTITUTION FUNCTION EXAMPLE

Substitution Function Medicare_report($Age) Applied to Test Cases					
Input Factors					**Equivalence Class Factor**
$Age	***$Weight***	***$Height***	***$Sex***	***$Intake***	***Medicare***
65	118	72	male	3000	Medicare_report($Age)
20	128	72	female	3000	Medicare_report($Age)
65	118	72	male	3000	yes
20	128	72	female	3000	no

PHP Function Medicare_report($Age)

```
<?php
function Medicare_report($Age) {
  if($Age>=65) return('yes');          # Medicare report is expected class
  if($Age>0) return('no');             # Medicare report is not expected
}
?>
```

III. COMBINATION FUNCTIONS FOR CONSTRAINTS

A. Four Functions for Four Constraints

This section shows how combination functions can implement constraints in combinatorial test designs. Four examples from Appendix D of [15] illustrate the ideas.

Constraint 1: (OS = "Windows") => (Browser = "IE" || Browser = "Firefox" || Browser = "Netscape").

The constraint says that if factor OS is Windows, then factor Browser must be IE, Firefox or Netscape. To be meaningful, the constraint needs another OS value, say Linux. For OS Linux, Browser must be Firefox or Netscape. The first panel of Table III shows a partition corresponding to the constraint in DPB notation. Two blocks are used, one for Windows and one for Linux. The second panel uses a determinant factor $OS and a function fBrowser($OS) in one block instead of fixed values. The third panel shows the PHP definition of the function. The function will be called for each $OS value, and it will return the allowed Browser values associated with the $OS value.

The fBrowser function does not return any values for disallowed $OS values. For example, if OS/360 is added to the list of $OS values, there will be no list of Browser values for OS/360. The function must return only allowed values to meet the DPB requirement that all combinations of factor values in a block are allowed. (This also means that every factor in a block must have at least one allowed value.)

Constraint 2: (P1 > 100) || (P2 > 100).

The constraint says that either factor P1 or P2 must be greater than 100. In this example there are 5 allowed values ranging from 89 to 103. The first panel of Table IV shows a partition corresponding to the constraint in DPB notation. Two blocks are used, one for each allowed condition. The second panel uses a determinant factor $P1 and a function fP2($P1) in one block instead of fixed values. The third panel shows the PHP definition of the function. The function will be called for each $P1 value, and it will return the allowed P2 values associated with the $P1 value.

TABLE III. CONSTRAINT 1 EXAMPLE

Constraint 1 Partition with Fixed Values
Constraint 1 without fBrowser function OS Browser # + Windows Windows IE Firefox Netscape + Linux Linux Firefox Netscape
Constraint 1 Partition with fBrowser($OS)
Constraint 1 with fBrowser function $OS Browser # Windows Linux fBrowser($OS)
PHP Function fBrowser($OS)
<?php function fBrowser($OS) { global $sp; # separator character switch($OS) { case 'Windows': $Browser='IE'.$sp.'Firefox'.$sp.'Netscape'; break; case 'Linux': $Browser='Firefox'.$sp.'Netscape'; break; } if(isset($Browser)) return($Browser); } ?>

Constraint 3: (P1 > P2) => (P3 > P4).

The constraint says that if factor P1 is greater than P2, then P3 must be greater than P4. The first panel of Table V shows a partition corresponding to the constraint in DPB notation. It uses 3 determinant factors $P1, $P2, $P3, and a function fP4($P1,$P2,$P3) in one block. The second panel shows the PHP definition of the function. The function will be called for each combination of the determinant values, and it will return the P4 values associated with the allowed combinations of $P1, $P2 and $P3.

Constraint 4: (P1 = true || P2 >= 100) => (P3 = "ABC").

The constraint says that if factor P1 is true, or if P2 is greater than or equal to 100, then P3 must be ABC. The first panel of Table VI shows a partition corresponding to the constraint in DPB notation. It uses 2 determinant factors $P1 and $P2, and a function fP3($P1,$P2) in one block. The second panel shows the PHP definition of the function. The function will be called for each combination of the determinant values, and it will return the allowed P3 values associated with the $P1, $P2 pairs.

TABLE IV. CONSTRAINT 2 EXAMPLE

Constraint 2 Partition with Fixed Values
Constraint 2 without fP2 function P1 P2 # +all P1 values with P2>100 89 97 100 101 103 101 103 +all P2 values with P1>100 101 103 89 97 100 101 103
Constraint 2 Partition with fP2($P1)
Constraint 2 with fP2 function $P1 P2 # 89 97 100 101 103 fP2($P1)
PHP Function fP2($P1)
<?php function fP2($P1) { global $sp; # separator character if($P1>100) $P2='89'.$sp.'97'.$sp.'100'.$sp.'101'.$sp.'103'; else $P2='101'.$sp.'103'; return($P2); } ?>

B. *Constraints for a Shopping Cart State Model*

This section shows how combination functions can simplify constraints significantly. The example is for an online shopping cart described in Section 6.4 of [15]. Figure 1 illustrates the shopping cart, which contains 3 different items. These items can be checked for deletion (and unchecked). New quantities to purchase can be entered. The update button effects these changes. The shop and checkout buttons take the user to the previous shopping page and the payment page respectively.

The test model is based on a UML state machine. Use of the state machine helps to associate input combinations with equivalence classes: Each target state represents an equivalence class. Constraints in the example include the following. Triggers must target one specific state to avoid masking; items in different positions must be different; and when there are empty positions in the cart, some factor values will not be applicable in the resulting test cases. Previously 33 blocks of fixed values defined the partition of this example. With embedded functions, the number of blocks can be reduced to 3 by using 7 simple combination functions (average length 7 lines).

Figure 2 shows a state diagram for the shopping cart. Transitions among the lowest-level (leaf) states are to be tested. The example focuses on transitions from the nonemptyCart state to the nonemptyCart state via the CHECK, QTY and UPDATE events. The partition is designed for one target state (one equivalence class), the nonemptyCart state, to avoid masking.

TABLE V. CONSTRAINT 3 EXAMPLE

Constraint 3 Partition with fP4($P1,$P2,$P3)

```
Constraint 3 with fP4 function
$P1
$P2
$P3
P4
#
89 97 100 101 103
89 97 100 101 103
89 97 100 101 103
fP4($P1,$P2,$P3)
```

PHP Function fP4($P1,$P2,$P3)

```
<?php
function fP4($P1,$P2,$P3) {
  global $sp;                              # separator character
  $return_values=FALSE;                    # nothing to return yet
  $P4_values=array(89,97,100,101,103);     # allowed $P4 values
  if($P1>$P2) {                            # constraint 3 antecedent
    foreach($P4_values as $P4) {           # try each $P4 value
      if($P3>$P4) {                        # constraint 3 consequent
        if(!$return_values) {              # this is the first value to return
          $P4_string=$P4;                  # start the string of values to return
          $return_values=TRUE;             # remember the string of values is started
        } else {                           # there are more than 1 values to return
          $P4_string.=$sp.$P4;             # append this value to the other(s) to return
        }
      }
      else continue;                       # this value is not allowed; try the next one
    }
  } else {                                 # all values are allowed; return them
    foreach($P4_values as $P4) {           # try each $P4 value
      if(!$return_values) {                # this is the first value to return
        $P4_string=$P4;                    # start the string of values to return
        $return_values=TRUE;               # remember the string of values is started
      } else {                             # there are more than 1 values to return
        $P4_string.=$sp.$P4;               # append this value to the other(s) to return
      }
    }
  }
  if($return_values) return($P4_string);   # if there are $P4 values, return them
}
?>
```

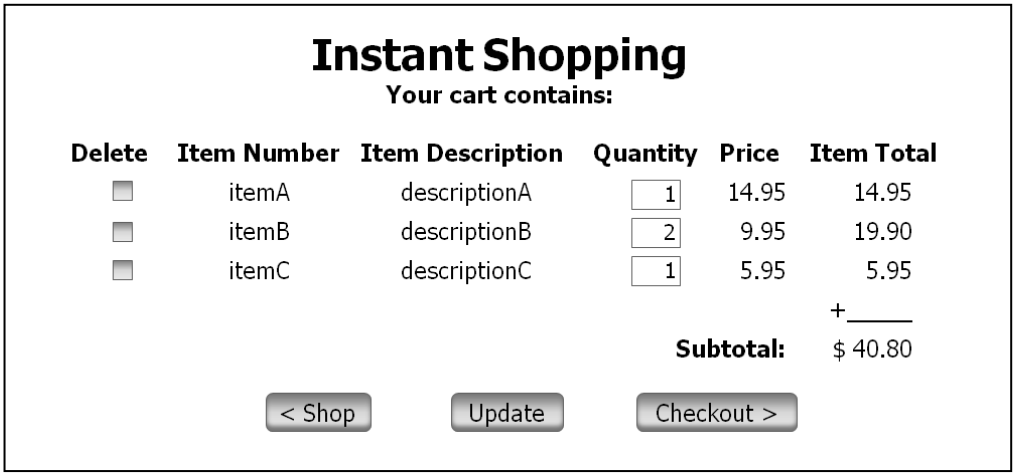

Fig. 1. Online shopping cart

TABLE VI. CONSTRAINT 4 EXAMPLE

Constraint 4 Partition with fP3($P1,$P2)
Constraint 4 with fP3 function $P1 $P2 P3 # true false 89 97 100 101 103 fP3($P1,$P2)
PHP Function fP3($P1,$P2)
<?php function fP3($P1,$P2) { global $sp; # separator character if($P1||$P2>=100) return('ABC'); else return('DEF'.$sp.'GHI'); } ?>

Table VII lists the test factors and their values for the shopping cart example. Where applicable, the combination functions returning values are given. An indication for the meaning of each test factor is also listed. Up to 3 different items are placed in the shopping cart for testing. Each is in a different cart position (0, 1 or 2), and the corresponding test factors have that index.

Some test factors can take the value NULL, which means the factor is not used or not applicable. The $newItem factor has the value NULL because there are no new items to place in the cart in this partition. In the nonemptyCart state there always is an item in position 0, so the corresponding values are not NULL. However when there are fewer than 3 different items in the cart, positions 1 and 2 may be unused. Then the corresponding values will be NULL.

TABLE VII. TEST FACTORS, VALUES AND FUNCTIONS FOR SHOPPING CART EXAMPLE

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

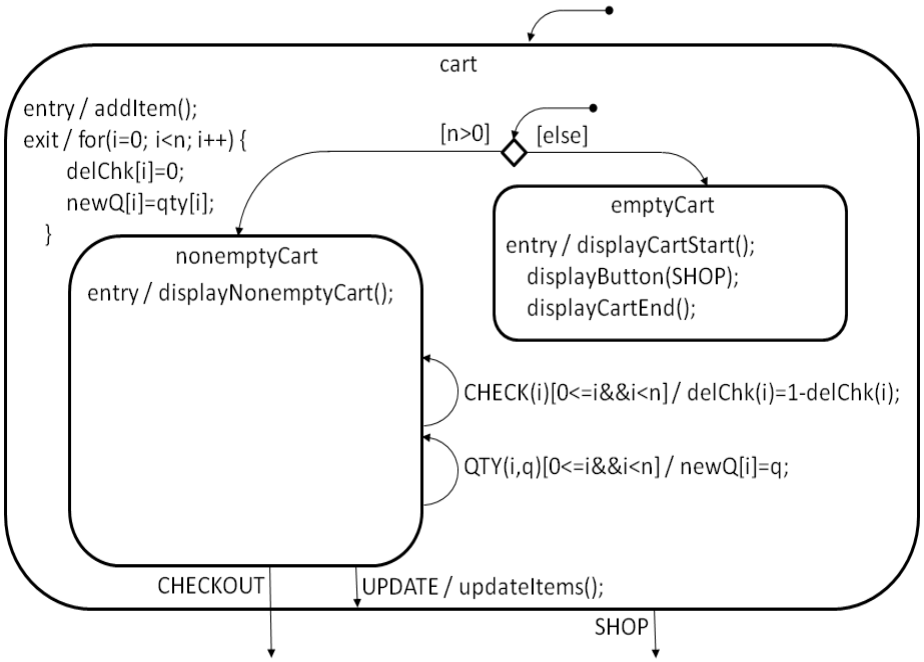

Fig. 2. State diagram for shopping cart

The 7 combination functions are listed in Table VII. In this model the same function is used for all the positions of the indexed factors. The functions are briefly described in the following list, and the PHP definitions for 3 of them are given in Table VIII.

- f_delChk($position,$n,$chkd1,$chkd2) returns values 0 (unchecked) and 1 (checked) for used positions (<$n); returns NULL for unused positions (>=$n); returns 0 if $chkd1 and $chkd2 are both 1, to constrain the transition to the nonemptyCart state after the UPDATE event (Table VIII).
- f_item($position,$n,$skip1,$skip2) in used positions returns items different from items in previous positions (<$position); returns NULL for unused positions (Table VIII).
- f_qty($position,$n) returns nonzero allowed values for current quantities in used positions; returns NULL for unused positions.
- f_newQ($position,$n,$zero1,$zero2) returns allowed values for new quantities selected by the QTY($i,$q) event; returns NULL for unused positions; returns nonzero quantity values if $zero1 and $zero2 are both 0, to constrain the transition to the nonemptyCart state after the UPDATE event (Table VIII).
- f_i($n) returns the used position values for the CHECK($i) and QTY($i,$q) events.
- f_event_CHECK($i) returns the event to check/uncheck the deletion box in position $i.
- f_event_QTY($i,$q) returns the event to select a new quantity $q for the quantity box in position $i.

Table IX shows the partition for this example using DPB notation (in 2 columns). There are 3 blocks, for the CHECK($i), QTY($i,$q) and UPDATE events respectively.

TABLE VIII. THREE OF THE SEVEN FUNCTIONS FOR THE SHOPPING CART EXAMPLE

PHP Functions f_delChk($position,$n,$chkd1,$chkd2) and f_newQ($position,$n,$zero1,$zero2)

```
<?php
function f_delChk($position,$n,$chkd1,$chkd2) {      # deletion box: 0 => not checked; 1 => is checked
 global $sp;                                         # separator character
 if($position<$n) {                                  # this position is used
  if($position==$n-1                                 # this is the last used position (<$n)
   &&$chkd1==1&&$chkd2==1) return('0');              # all prevous positions are checked for deletion, so return unchecked
                                                     #     for UPDATE to nonemptyCart (not to emptyCart)
  else return('0'.$sp.'1');                          # for other UPDATE cases, for CHECK & QTY, return 0 & 1
 }
 else return('NULL');                                # for unused positions return NULL
}
function f_item($position,$n,$skip1,$skip2) {        # items different from those in previous positions
 global $sp;                                         # separator character
 if($position>=$n) return('NULL');                   # return NULL for disallowed position
 $items=array('itemA','itemB','itemC');              # possible item values
 $item_count=0;
 foreach($items as $item) {                          # try each item
  if(strcmp($item,$skip1)==0) continue;              # skip this item to avoid duplicates in the cart
  if(strcmp($item,$skip2)==0) continue;              # skip this item to avoid duplicates in the cart
  if($item_count==0) $item_string=$item;             # start the string of values to return
  else $item_string.=$sp.$item;                      # append this value to the other(s) to return
  $item_count++;
 }
 return($item_string);                               # return the items for this position
}
function f_newQ($position,$n,$zero1,$zero2) {        # new quantity selected by the QTY event
 global $sp;                                         # separator character
 if($position<$n) {                                  # this position is used
  if($position==$n-1&&$zero1==0&&$zero2==0)          # this is the last used position; all prevous positions have selected quantities of 0
   return('1'.$sp.'2'.$sp.'10');                     # return nonzero quantities for UPDATE to nonemptyCart
  else return('0'.$sp.'1'.$sp.'2'.$sp.'10');         # for other UPDATE cases, for CHECK & QTY, 0 is allowed
 }
 else return('NULL');                                # for unused positions return NULL
}
?>
```

TABLE IX. PARTITION FOR SHOPPING CART EXAMPLE

nonemptyCart to nonemptyCart with PHP Functions	
Shopping Cart Example - transition design	+ nonemptyCart to nonemptyCart; QTY
$newItem	NULL
$n	1 2 3
$delChk[0]	f_delChk(0,$n,,)
$item[0]	f_item(0,$n,,)
$qty[0]	f_qty(0,$n)
$newQ[0]	f_newQ(0,$n,,)
$delChk[1]	f_delChk(1,$n,,)
$item[1]	f_item(1,$n,$item[0],)
$qty[1]	f_qty(1,$n)
$newQ[1]	f_newQ(1,$n,,)
$delChk[2]	f_delChk(2,$n,,)
$item[2]	f_item(2,$n,$item[0],$item[1])
$qty[2]	f_qty(2,$n)
$newQ[2]	f_newQ(2,$n,,)
$i	f_i($n)
$q	0 1 2 10
state	nonemptyCart
event	f_event_QTY($i,$q)
#NN nonemptyCart to nonemptyCart	
+ nonemptyCart to nonemptyCart; CHECK	+ nonemptyCart to nonemptyCart; UPDATE
NULL	NULL
1 2 3	1 2 3
f_delChk(0,$n,,)	f_delChk(0,$n,,)
f_item(0,$n,,)	f_item(0,$n,,)
f_qty(0,$n)	f_qty(0,$n)
f_newQ(0,$n,,)	f_newQ(0,$n,,)
f_delChk(1,$n,,)	f_delChk(1,$n,$delChk[0],1)
f_item(1,$n,$item[0],)	f_item(1,$n,$item[0],)
f_qty(1,$n)	f_qty(1,$n)
f_newQ(1,$n,,)	f_newQ(1,$n,$newQ[0],0)
f_delChk(2,$n,,)	f_delChk(2,$n,$delChk[0],$delChk[1])
f_item(2,$n,$item[0],$item[1])	f_item(2,$n,$item[0],$item[1])
f_qty(2,$n)	f_qty(2,$n)
f_newQ(2,$n,,)	f_newQ(2,$n,$newQ[0],$newQ[1])
f_i($n)	NULL
NULL	NULL
nonemptyCart	nonemptyCart
f_event_CHECK($i)	UPDATE

The first block specifies the allowed combinations for checking/unchecking the deletion boxes; the second block specifies those for selecting new quantities. These transitions are prior to an UPDATE event, so combinations which might empty the cart are included. The third block is for the UPDATE transition. Here the allowed combinations of the $delChk and $newQ factors constrain the transition to the nonemptyCart state as required.

In this example each of the functions f_event_CHECK($i) and f_event_QTY($i,$q) has a composite relationship with f_i($n). Each of them is a function of a function because the $i values depend on $n. Similar composite relations exist among other functions in the example.

IV. SUBSTITUTION FUNCTIONS FOR EQUIVALENCE CLASSES

This section illustrates how substitution functions can indicate the equivalence classes of test cases to help avoid masking. A body mass index (BMI) report application [5] provides the example. Requirements for the application are as follows.

R1. The listed input data will be stored in the patient database table.
   a. Age in years
   b. Weight in pounds
   c. Height in inches
   d. Sex (female, male)
   e. Intake in kilocalories per day

R2. The BMI will be calculated (in kilograms per meter squared) as $703.06957964 \text{ Weight} / \text{Height}^2$ and stored in the patient database table.

R3. If Age is 65 years or older, the Medicare report will be generated.

R4. If Age is younger than 20 years, the Child report containing the BMI percentile will be generated for the corresponding listed classification.
   a. Girl, from the female BMI-age table
   b. Boy, from the male BMI-age table

R5. If Age is 20 years or older, the Adult report will be generated for the corresponding listed classification.
   a. Underweight, BMI < 18.5
   b. Normal, $18.5 \leq$ BMI < 25.0
   c. Overweight, $25.0 \leq$ BMI < 30.0
   d. Obese, $30.0 \leq$ BMI

The BMI calculation is an expected result of R2. R3, R4, and R5 suggest three *equivalence class factors* which are functionally dependent on inputs:

Age	→ Medicare classes
Age, Sex	→ Child classes
Age, Weight, Height	→ Adult classes

The equivalence class factors characterize expected results, for analysis and verification. *Equivalence class functions* for these factors can be defined as in Table II for the Medicare classes, and in Table X for the Child classes and Adult classes. Thus,

- Medicare_report($Age) returns the expected Medicare classes;
- Child_report($Age,$Sex) returns the expected Child classes;
- Adult_report($Age,$Weight,$Height) returns the expected Adult classes.

When the 3 equivalence class factors are included in the design, using values from their respective substitution functions, the expected equivalence classes for each test case can be listed, as in Table XI.

Reference [5] provides formulas for coverage of the equivalence classes before test cases are generated. Evaluation of the equivalence class functions after test case generation enables a check of the coverage analysis and of the presence or absence of masking.

V. DISCUSSION

The embedded functions feature offers a new approach to the longstanding challenge of constraints and masking. The choice of a general-purpose language for these functions offers flexibility in their definitions that is suitable to a broad range of application domains. And the widespread knowledge and use of the language can facilitate the application of combinatorial testing to projects it can benefit.

In ongoing development programs, embedded functions can provide value that persists across product lines and releases. Engineers working in a particular application domain can create a function library to model systems under test and their constraints. These functions can be reused or adapted as needed, like other artifacts of the development program.

Development of the embedded functions feature is in progress. Assessment of the feature's feasibility, usability and performance will follow the implementation. With a positive assessment embedded functions will be deployed, to expand access to constraint handling and thus improve the practice of combinatorial testing.

TABLE X. SUBSTITUTION FUNCTIONS FOR BODY MASS INDEX EXAMPLE

PHP Functions Child_report($Age,$Sex), Adult_report($Age,$Weight,$Height) and BMI($Weight,$Height)

```
<?php
function Child_report($Age,$Sex) {
 if($Age>0) {
  switch($Sex) {
   case 'female':
    if($Age<20) return('girl');                  # Child report for girl is expected class
    else return('no');                           # Child report is not expected class
   case 'male':
    if($Age<20) return('boy');                   # Child report for boy is expected class
    else return('no');                           # Child report is not expected class
  }
 }
}
function Adult_report($Age,$Weight,$Height) {
 $bmi_value=BMI($Weight,$Height);                # Use BMI function to get BMI value
 if($Age>=20&&$bmi_value>0) {                    # Adult age and valid BMI value
  if($bmi_value>=30) return('obese');            # Adult report for obese is expected class
  if($bmi_value>=25) return('overweight');       # Adult report for overweight is expected class
  if($bmi_value>=18.5) return('normal');         # Adult report for normal is expected class
  if($bmi_value>0) return('underweight');        # Adult report for underweight is expected class
 }
 if($Age>0&&$bmi_value>0) return('no');          # Adult report is not expected class
}
function BMI($Weight,$Height) {                  # BMI function for Adult_report function
 if($Weight>0&&$Height>0) {                      # valid weight and height values
  $bmi_value=703.06957964*$Weight/$Height/$Height;
  return($bmi_value);                            # return valid BMI value
 }
}
?>
```

TABLE XI. TEST CASES AND CORRESPONDING EQUIVALENCE CLASS FACTOR VALUES

	Input factors					Equivalence class factors		
	Age	***Weight***	***Height***	***Sex***	***Intake***	***Medicare***	***Child***	***Adult***
gl1	15	115	63	female	2000	no	girl	no
gl2	19	135	67	female	2000	no	girl	no
gl3	19	135	63	female	3000	no	girl	no
gl4	19	115	67	female	3000	no	girl	no
gl5	15	135	67	female	3000	no	girl	no
by1	15	125	66	male	2000	no	boy	no
by2	19	168	71	male	2000	no	boy	no
by3	19	168	66	male	3000	no	boy	no
by4	19	125	71	male	3000	no	boy	no
by5	15	168	71	male	3000	no	boy	no
uw1	65	118	72	male	3000	yes	no	underweight
uw2	65	128	70	female	2000	yes	no	underweight
uw3	64	128	72	male	2000	no	no	underweight
uw4	20	118	70	male	2000	no	no	underweight
uw5	64	118	70	female	3000	no	no	underweight
uw6	20	128	72	female	3000	no	no	underweight
nm1	65	129	70	male	3000	yes	no	normal
nm2	65	154	66	female	2000	yes	no	normal
nm3	64	154	70	male	2000	no	no	normal
nm4	20	129	66	male	2000	no	no	normal
nm5	64	129	66	female	3000	no	no	normal
nm6	20	154	70	female	3000	no	no	normal
ow1	65	155	66	male	3000	yes	no	overweight
ow2	65	174	64	female	2000	yes	no	overweight
ow3	64	174	66	male	2000	no	no	overweight
ow4	20	155	64	male	2000	no	no	overweight
ow5	64	155	64	female	3000	no	no	overweight
ow6	20	174	66	female	3000	no	no	overweight
ob1	65	175	64	male	3000	yes	no	obese
ob2	65	211	61	female	2000	yes	no	obese
ob3	64	211	64	male	2000	no	no	obese
ob4	20	175	61	male	2000	no	no	obese
ob5	64	175	61	female	3000	no	no	obese
ob6	20	211	64	female	3000	no	no	obese

REFERENCES

[1] K. Tatsumi, "Test case design support system," Proceedings of the International Conference on Quality Control, Tokyo: 1987, pp. 615-620.

[2] G. B. Sherwood, "Improving test case selection with constrained arrays," AT&T Bell Labs Technical Memorandum, March 26 1990, unpublished. Retrieved January 8, 2015, from Testcover.com: http://testcover.com/pub/background.

[3] G. B. Sherwood, "Effective testing of factor combinations," Third International Conference on Software Testing, Analysis & Review, Washington, DC: May 1994, pp. 151-166.

[4] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios. Proceedings of the Twenty-fourth Annual Pacific Northwest Software Quality Conference, Portland, OR: 2006, pp. 419-430.

[5] G. B. Sherwood, "Functional dependence and equivalence class factors in combinatorial test designs," Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, Cleveland, OH: 2014, pp. 108-117.

[6] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, Greenbelt, MD: 2006, pp.153-158.

[7] M. B. Cohen, M. B. Dwyer and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," Proceedings of the 2007 International Symposium on Software Testing and Analysis, London: 2007, pp. 129-139.

[8] A. Calvagna and A. Gargantini, "A logic-based approach to combinatorial testing with constraints," Tests and Proofs, Proceedings of the Second International Conference, Prato, Italy: 2008, pp. 66-83.

[9] M. B. Cohen, M. B. Dwyer and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach," IEEE Transactions on Software Engineering, vol. 34, pp. 633-650, 2008.

[10] A. Calvagna and A. Gargantini, "Combining satisfiability solving and heuristics to constrained combinatorial interaction testing," Tests and Proofs, Proceedings of the Third International Conference, Zurich: 2009, pp. 27-42.

[11] A. Calvagna and A. Gargantini, "A formal logic approach to constrained combinatorial testing," Journal of Automated Reasoning, vol. 45, pp. 331-358, 2010.

[12] L. Li, Y. Cui, Y. Yang, "Combinatorial test cases with constraints in software systems," Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design, Wuhan, China: 2012, pp. 195-199.

[13] C. D. Nguyen, A. Marchetto and, P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," Proceedings of the 2012 International Symposium on Software Testing and Analysis, Minneapolis: 2012, pp. 100-110.

[14] M. Achour, F. Betz, A. Dovgal, et al., PHP Manual. Retrieved January 8, 2015, from the PHP Group: http://php.net/manual/en/index.php.

[15] D. R. Kuhn, R. N. Kacker and Y. Lei, Introduction to Combinatorial Testing, CRC Press, Boca Raton, FL: 2013.