Embedded Functions for Constraints and Variable Strength in Combinatorial Testing

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

***Abstract*—An embedded functions feature was implemented to specify functionally dependent relations among test factors. Functions embedded in a combinatorial test case generator specified test factor constraints to which the resulting test cases conformed. The functions were defined in a general-purpose programming language widely used among software engineers. Examples with and without embedded functions were compared. Embedded functions were used to evaluate equivalence class factors to insure coverage of selected classes of results. Embedded functions also were used to evaluate hybrid factors in variable strength designs. Usability and performance characteristics were described.**

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

I. INTRODUCTION

Combinatorial testing generates relatively small sets of test cases for complex systems. Such a system has k test factors (e.g. test configuration choices and input parameters). The systematic application of combinations of test factor values (t-tuples, with $t < k$) significantly reduces the number of test cases versus those of all combinations. These *strength t* test cases can result in more effective testing and higher quality compared with other test case selection methods.

This work reports on a project to improve the usability and efficiency of conforming to system constraints in combinatorial testing. Functionally dependent relations appear as constraints in test models. A test factor is *functionally dependent* when its value is identified by those of its *determinant factors*. E.g., to test using a date input, the value of last_day(month,year) may be needed to verify month boundaries. Reference [1] proposed using *embedded functions* (EF) in a combinatorial test tool to specify test constraints in a language familiar to software engineers. Embedded functions, like last_day above, would return values for dependent factors based on the values of determinant factors. The relations among the factor values would define the constraints for the system under test.

Test constraints can arise from: (1) test configuration requirements, (2) test input requirements, and (3) coverage requirements for classes of expected results. Consequences of test plans that do not conform to constraints may include: (1) impossible or unsupported configurations, (2) inputs that cannot be entered or that lead to error cases, and (3) insufficient coverage (missing t-tuples) for classes of results.

All of these situations can lead to *masking* [2,3,4], which reduces the benefit of t-tuple coverage. Masking occurs when a required t_m-tuple ($t_m \leq t$) is missing from the test cases for a class of results. When test cases (k-tuples) cannot be run, t_m-tuples can be skipped. When t-tuples are distributed among disjoint classes of results, some classes can miss expected t-tuples. Conformance to system constraints is essential for effective combinatorial testing. Thus, improved usability for constraint conformance is a powerful motivation for embedded functions.

This paper describes an initial implementation of the embedded functions feature which was introduced in [1]. Examples compare the use of embedded functions with that of manually selected, fixed factor values. Examples requiring composite relations to describe complex constraints are shown. Embedded functions enable the association of values from 2 or more factors. An immediate consequence is the possibility of generating subarrays of higher strength, i.e. variable strength designs with $t < t'$ and $k' < k$ factors [5]. The paper offers some higher strength schemes and reports on performance characteristics of the implementation.

Overall objectives for the work were as follows.

1. *Specify constraints among test factors with simple functions in an established programming language.*
2. *Evaluate composite, embedded functions automatically to generate test cases conforming to test model constraints.*
3. *Generate variable strength designs using embedded functions.*
4. *Generate test case designs with suitable response times.*
5. *Generate test case designs with practical sizes.*

The remainder of the paper consists of the following sections, II. General Methods, III. Strength 2 Results, IV. Variable Strength Methods, V. Variable Strength Results, VI. Discussion. Details for all of the examples of this paper are available at Testcover.com [6]. Test case generation requests, results and embedded function listings are posted there for review.

II. GENERAL METHODS

A. User Interface

PHP [7] was the language selected for embedded functions in this project. PHP is broadly used, and it supports both user-defined and built-in functions. The embedded functions feature enhanced Direct Product Block (DPB) notation to accept factor values returned from PHP functions. (DPB notation was

/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSTW.2016.16

introduced in Section 6.1.3 of [8].) The test case generation request is entered in DPB notation, in a text box on the generator form. A summary of DPB notation is presented in Table I. The allowed values for each factor appear on a separate line in a block. All combinations of factor values in the block are allowed. Partitions are sets of blocks. A partition's allowed combinations include all of its blocks' combinations, and no disallowed combinations.

The first panel of Table II gives an example of DPB notation to specify valid dates according to calendar rules. Manually selected fixed values (FV) specify the constraints in 5 *FV blocks*. The first block specifies days 1 and 10 for all months and 3 years; the other 4 blocks include the last day for each month. In the enhanced DPB notation, functions can be listed as values for dependent factors. In the second panel of Table II, 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 [7] and start with a letter or underscore (_). The arguments to the function are the determinant factor names $month and $year. Determinant factor names are PHP variables, so they are represented by a dollar sign ($) followed by the name of the variable. In the second panel, the Day factor is functionally dependent (FD), so it is contained in an *FD block*.

The first two panels of Table II show that the last_day function allows the number of blocks in the request 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 [7] can support a broader range of years.

TABLE I. DPB NOTATION SUMMARY

A *request* contains the data for one set of test factors. It is a multi-line string consisting of an optional heading and 1 or more partitions.

A *heading* is a string of 1 or more lines consisting of a request_name and optional factor_names.

A *factor_name* labels a column of factor values in the results, e.g. Browser, $month, $a. A factor_name that is a PHP variable (starting with $) can label a determinant factor.

A *partition* is a multi-line string containing 1 or more blocks. It represents all the allowed combinations of factor values for 1 test case generation instance.

A *block* contains 2 or more factor_value_lines. All combinations of factor values in a block are allowed. The union of all the blocks' combinations defines their partition.

A *factor_values_line* is a string containing 1 or more factor values; these can be fixed values or embedded functions, e.g. 1 10 last_day($month,$year). A dependent factor has 1 or more functions among its values.

An *embedded function* returns a string of 1 or more fixed values; the function is 1 of:
a PHP user-defined function, e.g. last_day($month,$year)
a PHP internal (built-in) function, e.g. max($a,100)
or a determinant factor_name, e.g. $a

A function argument is either a determinant factor_name or a fixed value.

A *fixed value* is either: a Boolean, e.g. TRUE, or a string, e.g. 100

Types may be changed by context during evaluation.

TABLE II. EMBEDDED 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 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. Request Processing

Examples described here were run on a Testcover.com server configured for production, using Intel Xeon ES-2650 2.00 GHz processors. The server ran Linux 6.7, Apache/2.2.15 and PHP 5.3.3. Requests were submitted using a browser running on the server (Firefox ESR). The response time for each request was taken as the difference between its start and stop log entries.

The embedded functions enhancement was implemented in the program which processes test case generation requests. An automated process converted FD blocks to FV blocks, based on the procedure of [9]. The automated process also contained proprietary enhancements to limit the number of FV blocks generated.

The program parsed the factor values to find the names and arguments of embedded functions. To process composite functions, the program ordered the functions of each FD block for evaluation. The ordering insured that each function's arguments were fixed when it was evaluated.

Each function was evaluated for all combinations of its determinant factors. The resulting function values were used to generate 1 or more blocks having fixed values to replace the function of the original FD block. Each function's evaluation used the blocks of the previous evaluation, so that after all the evaluations were done, the generated blocks were in FV form.

With the block from the original request in FV form, the service processed it as usual, with other FD blocks in the partition converted to FV blocks as needed. At times there were too many blocks in a partition to process all at once. When this happened the blocks were automatically grouped into bundles for sequential processing.

C. Test Models

1) Simple constraint models

The embedded functions feature was used with 6 simple test models to illustrate test case generation in conformance with constraints. Each of the first 5 examples used 1 dependent factor having values from 1 function in a single FD block. The examples were the Calendar example above and 4 constraints examples (in Table III) from Appendix D of [8]. Descriptions of the examples, their test case generation requests and embedded function listings were given in Section III of [1].

Reference [10] introduced FD equivalence class factors, which indicate classes of expected results. In a test design equivalence class factors can insure selected classes are covered. Without equivalence class factors, the Body Mass Index (BMI) example (Table X in [10]), required a strength 4 design with 24 test cases to associate each class of results with its nondeterminant factor values in 1 partition [10]. Here the BMI example used 3 embedded functions in a strength 2 design in 1 partition. The goal was to reach the selected classes of results and to associate them with their nondeterminant factor values.

2) Shopping cart models

The primary example was an online shopping cart described in [1] and Section 6.4 of [8]. Additional details and related models were given in [11]. A summary of the model is recapped in this section to convey the methods and to clarify the few differences between the model proposed [1] and the one used here.

Fig. 1 illustrates the shopping cart, containing 3 different items. The items can be checked for deletion (and unchecked). New quantities 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.

TABLE III. FOUR SIMPLE CONSTRAINTS

Example	Constraint
Constraint 1	(OS = "Windows") => (Browser = "IE" || Browser = "Firefox" || Browser = "Netscape")
Constraint 2	(P1 > 100) || (P2 > 100)
Constraint 3	(P1 > P2) => (P3 > P4)
Constraint 4	(P1 = true || P2 >= 100) => (P3 = "ABC").

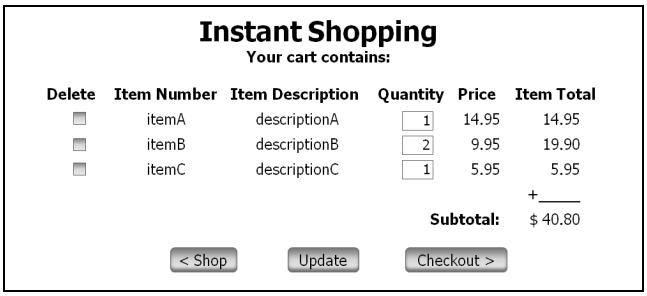

Fig. 1. Online shopping cart

The test model was based on a UML state machine. The state machine helped to associate input combinations with equivalence classes: Each target state represented an equivalence class. Constraints in the example included the following. Triggers target one specific state to avoid masking; items in different positions must be different; and when there are empty positions in the cart, their factor values will not be applicable in the test cases. Previously 33 manually selected FV blocks defined the partition of this example. With embedded functions, the number of (FD) blocks was reduced to 3 by using 9 simple functions.

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

Table IV lists the test factors and their values for the shopping cart example. Where applicable, the 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.

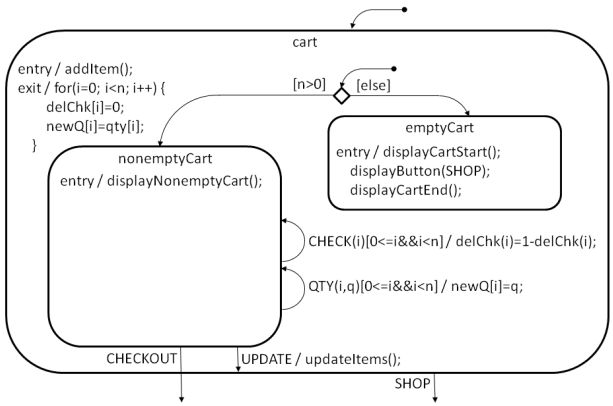

Fig. 2. State diagram for shopping cart

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

Test Factor	Test Factor Values	Functions	Indication
$newItem	NULL		Item to place in cart
$n	1 2 3		Number of items in cart
$delChk[0]	0 1	f_delChk_CQ f_delChk_U	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_CQ f_newQ_U	New quantity shown in cart position 0
$delChk[1]	0 1 NULL	f_delChk_CQ f_delChk_U	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_CQ f_newQ_U	New quantity shown in cart position 1
$delChk[2]	0 1 NULL	f_delChk_CQ f_delChk_U	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_CQ f_newQ_U	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

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 are NULL.

The 9 functions are listed in Table IV. In this model the same function was used for all the positions of the indexed factors in each block. The functions are briefly described in the following list, and their PHP definitions are given in [6].

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

Table V 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.

Compared to the earlier model [1], functions f_delChk_CQ and f_delChk_U replaced f_delChk, and functions f_newQ_CQ and f_newQ_U replaced f_newQ. These changes made 2 improvements. First, use of f_delChk_CQ and f_newQ_CQ in the CHECK and QTY blocks allowed for all items to be marked for deletion and for their selected quantities to be zero when the event was not UPDATE. That is, the target state remained nonemptyCart. Second, both f_delChk_U and f_newQ_U used $delChk and $newQ values for previous items to insure that the UPDATE event targeted the nonemptyCart state.

III. STRENGTH 2 RESULTS

Table VI presents a summary of the strength 2 results. The FV examples used manually selected fixed values; the EF examples used automatically evaluated embedded functions. The table lists the corresponding array parameters. The blocks column gives the number of FV blocks, either manually selected (for the FV examples), or automatically generated (for the EF examples). The combinations column shows the numbers of pairwise combinations. The total depends on the numbers of values for all factors; the residue is the number not covered due to constraints. The response time for each request is the difference between its start and stop log entries. Consequently fractions of seconds were truncated.

The Calendar, Constraint 1 and Constraint 2 examples (Tables II and III) were run both with manually selected fixed values and with embedded functions. Each FV example used the number of FV blocks given in Table VI; each EF example used 1 functionally dependent block which led to the same number of FV blocks after evaluation. Corresponding FV and EF examples covered the same numbers of combinations and generated the same test cases.

In the Constraint 3 EF example the values for factor P4 were given by the function fP4, which depended on the values of 3 determinant factors $P1, $P2, $P3. From 1 FD block 21 FV blocks were generated automatically; they led to 37 test cases. All pairwise combinations were covered while conforming to Constraint 3.

TABLE V. 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
$delChk[0]	f_delChk_CQ(0,$n)
$item[0]	f_item(0,$n,,)
$qty[0]	f_qty(0,$n)
$newQ[0]	f_newQ_CQ(0,$n)
$delChk[1]	f_delChk_CQ(1,$n)
$item[1]	f_item(1,$n,$item[0],)
$qty[1]	f_qty(1,$n)
$newQ[1]	f_newQ_CQ(1,$n)
$delChk[2]	f_delChk_CQ(2,$n)
$item[2]	f_item(2,$n,$item[0],$item[1])
$qty[2]	f_qty(2,$n)
$newQ[2]	f_newQ_CQ(2,$n)
$i	f_i($n)
$q	0 1 2 10
state	nonemptyCart
event	f_event_QTY($i,$q)
#NN nonemptyCart to nonemptyCart	
+ nonemptyCart to nonemptyCart; CHECK	+ nonemptyCart to nonemptyCart; UPDATE
NULL	NULL
1 2 3	1 2 3
f_delChk_CQ(0,$n)	f_delChk_U(0,$n,1,0,1,0)
f_item(0,$n,,)	f_item(0,$n,,)
f_qty(0,$n)	f_qty(0,$n)
f_newQ_CQ(0,$n)	f_newQ_U(0,$n,1,0,1,0)
f_delChk_CQ(1,$n)	f_delChk_U(1,$n,$delChk[0],$newQ[0],1,0)
f_item(1,$n,$item[0],)	f_item(1,$n,$item[0],)
f_qty(1,$n)	f_qty(1,$n)
f_newQ_CQ(1,$n)	f_newQ_U(1,$n,$delChk[0],$newQ[0],1,0)
f_delChk_CQ(2,$n)	f_delChk_U(2,$n,$delChk[0],$newQ[0],$delChk[1],$newQ[1])
f_item(2,$n,$item[0],$item[1])	f_item(2,$n,$item[0],$item[1])
f_qty(2,$n)	f_qty(2,$n)
f_newQ_CQ(2,$n)	f_newQ_U(2,$n,$delChk[0],$newQ[0],$delChk[1],$newQ[1])
f_i($n)	NULL
NULL	NULL
nonemptyCart	nonemptyCart
f_event_CHECK($i)	UPDATE

In the Constraint 4 EF example the values for factor P3 were given by the function fP3, which depended on the values of 2 determinant factors $P1, $P2. In this example the values for $P1 were interpreted as the Boolean constants TRUE and FALSE. From 1 FD block 3 FV blocks were generated; they led to 12 test cases.

The BMI EF example used 1 FD block with 5 test factors and 3 equivalence class factors. Each of the Medicare, Child and Adult equivalence class factors used 1 function to identify the classes of its expected results. Each function was evaluated for all combinations of its determinant factor values, so all allowed classes were included in the equivalence class factor's values. Each of these classes was paired with all allowed test factor values. The 14 test cases reached 10 equivalence classes and covered with nondeterminant strength 2 [10].

The Shopping cart FV and EF examples were very similar, but differences in their test models prevented identical results. The FV example used manually selected blocks. To limit their number to 33, not all permutations of itemA, itemB and itemC occurred in the n positions: Position 0 had any of the 3 items; position 1 had itemB or itemC; position 2 had only itemC. The simplification was justified because all items were equivalent; what mattered in the example was that different items were in different positions. This simplification was unnecessary in the Shopping cart EF example; the embedded functions allowed for all permutations of items in $n = 1, 2 or 3 positions by generating more blocks.

The Shopping cart EF example used 3 FD blocks, 1 for each of the CHECK, QTY and UPDATE events. The blocks required evaluation of 14, 14 and 12 instances of embedded functions respectively.

The order of evaluation for each block turned out to be the order given in the request, shown in Table V. However the ordering was necessary because there were several composite relations among the functions, as shown in Table V. For example, in the case of function f_item, when there were $n = 3 items, value(s) returned depended on earlier instances:

f_item: (0,$n,,) → $item[0]
f_item: (1,$n,$item[0],) → $item[1]
f_item: (2,$n,$item[0],$item[1]) → $item[2]

TABLE VI. STRENGTH 2 RESULTS

Example	Array parameters ($N;t,k,v_1^{k_1}\ldots v_s^{k_s}$)	Blocks	Combinations Total:Residue	Response time HH:MM:SS
Calendar FV	$(40;2,3,12^1 6^1 3^1)$	5	126:38	00:00:01
Calendar EF	$(40;2,3,12^1 6^1 3^1)$	5	126:38	00:00:01
Constraint 1 FV	$(5;2,2,3^1 2^1)$	2	6:1	00:00:01
Constraint 1 EF	$(5;2,2,3^1 2^1)$	2	6:1	00:00:00
Constraint 2 FV	$(16;2,2,5^2)$	2	25:9	00:00:01
Constraint 2 EF	$(16;2,2,5^2)$	2	25:9	00:00:00
Constraint 3 EF	$(37;2,4,5^4)$	21	150:0	00:00:01
Constraint 4 EF	$(12;2,3,5^1 3^1 2^1)$	3	31:8	00:00:00
Body Mass Index EF	$(14;2,8, 5^1 3^2 2^5)$	10	189:36	00:00:01
Shopping cart FV	$(91;2,18,16^1 5^3 4^4 3^6 2^2 1^2)$	33	2291:431	00:00:02
Shopping cart EF	$(98;2,18,16^1 5^3 4^6 3^5 2^1 1^2)$	183	2499:383	00:00:09

In the Shopping cart FV example the 33 manually selected blocks were distributed as 6 for the CHECK event, 24 for the QTY event and 3 for the UPDATE event. In the Shopping cart EF example 183 FV blocks were generated, 25, 100 and 58 respectively for the CHECK, QTY and UPDATE events. Compared to the FV example, the EF example generated 8% more test cases while covering 14% more combinations.

IV. VARIABLE STRENGTH METHODS

Often combinatorial test designs use a fixed strength t to define the coverage among test factor values. All allowed t-tuples among the k test factors are required to be covered. As t increases the coverage improves, and the cost of testing (e.g. number of test cases) increases. Variable strength [5] adds flexibility to cover a subset of k' test factors with higher strength t', while the rest of the factors retain strength-t coverage. Variable strength designs can enable additional test coverage in areas of particular importance or risk. Thus it offers cost-benefit choices beyond those of fixed strength alone. This section describes methods for variable strength designs using functionally dependent hybrid factors.

A. Functionally Dependent Hybrid Factors

A hybrid factor represents all the allowed combinations of values among 2 or more test factors. Williams and Probert used a hybrid factor to replace two interdependent test factors with 1 factor independent of the others [12]. Here embedded functions are used to handle such constraints. But including hybrid factors in a test design enables associations among combinations of factors (represented by the hybrid factors) and other individual factors.

Consider k test factors named \$a, \$b, \$c, … and the function pair(\$a,\$b), shown in Table VII, which returns the 2-tuple (\$a,\$b) when called. Table VIII illustrates a design for $k = 8$ test factors. Factor \$b has 3 values; all the other test factors have 2. Two functionally dependent factors, named (\$a,\$b) and (\$c,\$d), are included in the design (for a total of $K = 10$ factors). Factors (\$a,\$b) and (\$c,\$d) have values given by the functions pair(\$a,\$b) and pair(\$c,\$d), and thus represent the respective pairs. When all pairs of (\$a,\$b) and (\$c,\$d) are covered, the subarray consisting of factors \$a \$b \$c \$d is covered with strength $t' = 4$.

TABLE VII. PAIRING FUNCTION TO COMBINE TWO FACTORS

PHP Function pair(\$a,\$b)

```
<?php
function pair($a,$b) {
  return('('.$a.','.$b.')');    # return the pair of arguments given
}
?>
```

The subarray of \$a \$b \$c \$d is called the *nominal subarray* to distinguish it from other higher strength subarrays that may result from the hybrid factors. In this formulation the nominal subarray has k' test factors and strength t', with $t \le t' \le k' \le k$. Examination of Table VIII shows that all 24 combinations of \$a, \$b, \$c and \$d are included in test cases. Moreover, all 12 combinations of \$a, \$b and any other test factor are covered; all combinations of \$c, \$d and any other test factor are covered also.

In this example there are no constraints among the original test factors. That characteristic is not changed by the inclusion of the pairing factors. But the resulting array does have constraints. For example, the value \$a = 1 cannot be associated with the pair (\$a,\$b) = (0,0).

Using pairing functions for variable strength is compatible with constraints among test factors. Specifically, when a pairing function factor is appended to an FD block, it insures that all pairs of \$a and \$b are represented by its (\$a,\$b) values in that block and in the subsequent FV blocks. And if the values of \$a and \$b include functions of other test factors, the support for composite functions enables the design to follow the corresponding constraints. When the pairing function appears in all the blocks of a partition, it results in a *pairing factor*, which conforms to the constraints implied by the set of all combinations of the FV blocks.

TABLE VIII. VARIABLE STRENGTH TEST CASES

	\$a	\$b	\$c	\$d	\$e	\$f	\$g	\$h	(\$a,\$b)	(\$c,\$d)
1	1	1	1	1	0	0	0	0	(1,1)	(1,1)
2	0	2	0	0	1	1	1	0	(0,2)	(0,0)
3	1	0	1	0	1	1	0	1	(1,0)	(1,0)
4	0	1	0	1	1	0	1	1	(0,1)	(0,1)
5	0	0	1	1	0	1	1	1	(0,0)	(1,1)
6	1	2	0	1	0	0	0	0	(1,2)	(0,1)
7	0	0	0	0	0	0	0	0	(0,0)	(0,0)
8	1	1	0	0	0	1	1	1	(1,1)	(0,0)
9	1	2	1	0	1	0	1	1	(1,2)	(1,0)
10	0	1	1	0	0	0	0	0	(0,1)	(1,0)
11	1	0	0	1	0	0	0	0	(1,0)	(0,1)
12	0	2	1	1	0	0	0	0	(0,2)	(1,1)
13	1	1	0	1	1	1	1	0	(1,1)	(0,1)
14	1	2	1	1	1	1	1	0	(1,2)	(1,1)
15	1	0	1	1	1	1	1	0	(1,0)	(1,1)
16	0	2	1	0	1	1	0	1	(0,2)	(1,0)
17	0	1	1	1	1	1	1	0	(0,1)	(1,1)
18	0	0	1	0	1	1	1	0	(0,0)	(1,0)
19	1	1	1	0	0	0	0	0	(1,1)	(1,0)
20	1	2	0	0	0	0	0	0	(1,2)	(0,0)
21	1	0	0	0	0	0	0	0	(1,0)	(0,0)
22	0	2	0	1	0	0	0	0	(0,2)	(0,1)
23	0	1	0	0	0	0	0	0	(0,1)	(0,0)
24	0	0	0	1	0	0	0	0	(0,0)	(0,1)

A similar tripling function, triple($a,$b,$c), can be used to append a *tripling factor* to the design, insuring that all allowed combinations of $a, $b and $c are associated with the values of all the other k-3 test factors. These hybrid factors, individually and in combination, can be used to construct variable strength designs that apply additional coverage as needed.

B. Pairing Factors

This section describes pairing factors for subarrays with $t' \le 4$ and $k' \le 6$. Examples are given in Table IX.

1) Strength 3 subarrays

Consider p test factors $a, $b, $c, … which are to be covered with strength $t' = 3$. Assign them to 2 disjoint sets of factors. Because there are only 2 sets, any combination of 3 test factors must have at least 2 in one set or the other. For each pair of factors in a set, include a pairing factor in the test design. Each value of each pairing factor will be associated with each individual factor's values, so all allowed 3-tuples will be covered.

For example, with $p = 5$, assign $a and $b to the first set, and $c, $d and $e to the second. Include pairing factor ($a,$b) from the first set, and ($c,$d), ($c,$e) and ($d,$e) from the second. Each of the ($a,$b) pairs has a test case with each value of $c, $d and $e. Similarly, each pair from the second set, e.g. ($c,$d), is associated with $a and $b. This ($c,$d) pair is associated with $e as well. All the pairs are associated with the other individual factors, so these 4 pairing factors insure $t' = 3$ coverage of the 5 test factors.

Generally q factors can be chosen for the first set and the remaining p-q for the second. The total number of pairing factors $C(q,2) + C(p\text{-}q,2)$ is minimized when $q = p/2$ if p is even, or when $q = (p\text{-}1)/2$ if p is odd; the total number of pairing factors is $p(p\text{-}2)/4$ or $(p\text{-}1)^2/4$ respectively. In Table IX the subarrays following this construction are marked with an asterisk (*). In the $(t',k') = (2,2)$ case, $p = 3$, and the strength-3 subarray includes a test factor outside the nominal subarray, which includes factors $a and $b. In the other cases $p = k'$. When $p = 4$, the nominal subarray covers with $t' = 4$.

2) Strength 4 subarrays

To cover k' test factors with strength 4, pairs of pairing factors can be used. For $k' = 4$, the 2 pairing factors ($a,$b) and ($c,$d) associate all combinations of the 4 factors, as shown in Table VIII. Typically a subset of the $C(k',2)$ factor pairs is sufficient to cover all combinations of 4 factors. In the $(t',k') = (4,5)$ case (Table IX), when the pairs of pairing factors include distinct individual factors (e.g. ($a,$b) with ($d,$e)), 4 of the 5 factors are associated. There are 5 such pairs of pairing factors, and each covers a different combination of 4 test factors. Thus the 5 pairing factors insure the subarray covers with strength $t' = 4$. In the $k' = 6$ example, only 10 of the 15 possible pairing factors are needed.

C. Tripling Factors

Examples of tripling and pairing factors for subarrays with $t' \le 6$ and $k' \le 6$ are given in Table IX. As mentioned above, a hybrid factor using a tripling function, e.g. triple($a,$b,$c), may be used to cover 4-tuples. A subarray of 5 test factors can be covered with strength 5 using 1 tripling factor and 1 pairing factor containing 5 distinct test factors: All allowed triples and pairs are combined. In the $(t',k') = (5,6)$ case, 4 tripling factors containing the 6 test factors enable strength 5 coverage: Each of the 6 pairs of tripling factors combines 5 of the 6 test factors. And finally, 2 tripling factors containing 6 distinct test factors cover 6 test factors with strength 6.

D. Associated Subarrays

The coverage due to hybrid factors extends beyond the nominal subarray in variable strength designs. There are associated subarrays of higher strength which contain some or all of the nominal subarray factors. Their effect is to extend the higher strength associations beyond the nominal subarray. For example, in the $(t',k') = (5,5)$ case, there are $C(5,3)$ strength 3 arrays in the nominal subarray. And each of the $C(3,2) + C(2,2)$ pairs from the hybrid factors is paired with k-5 other, individual test factors. So there is a total of $4k$-10 strength 3 subarrays due to the 2 hybrid factors. Similarly, the tripling factor leads to k-3 strength 4 subarrays. Numbers of associated subarrays of strength 3 and 4, $n^{(3)}$ and $n^{(4)}$, are given for the examples in Table IX.

V. Variable Strength Results

A. Examples without Test Factor Constraints

Table X presents results for the 10 variable strength examples of Table IX, with $k = 8$ test factors. The results without hybrid factors, VS (2,8), are shown also. Details of these results are available in [6].

TABLE IX. Hybrid Factors for Higher Strength

Factors of nominal subarray		(t',k')	Hybrid pairing/tripling factors	Associated subarrays $n^{(3)}$	$n^{(4)}$
$a $b	*	(2,2)	($a,$b)	k-2	
$a $b $c		(3,3)	($a,$b,$c)	3k-8	k-3
$a $b $c $d	*	(4,4)	($a,$b) ($c,$d)	2k-4	1
$a $b $c $d $e	*	(3,5)	($a,$b) ($c,$d) ($c,$e) ($d,$e)	4k-10	3
		(4,5)	($a,$b) ($a,$e) ($b,$c) ($c,$d) ($d,$e)	5k-15	5
		(5,5)	($a,$b,$c) ($d,$e)	4k-10	k-3
$a $b $c $d $e $f	*	(3,6)	($a,$b) ($a,$c) ($b,$c) ($d,$e) ($d,$f) ($e,$f)	6k-16	9
		(4,6)	($a,$b) ($a,$c) ($a,$f) ($b,$c) ($b,$d) ($c,$d) ($c,$e) ($d,$e) ($d, $f) ($e,$f)	10k-40	15
		(5,6)	($a,$b,$c) ($a,$d,$e) ($b,$d,$f) ($c,$e,$f)	12k-52	4k-9
		(6,6)	($a,$b,$c) ($d,$e,$f)	6k-16	2k+3

As in the example of Table VIII, factor $b has 3 values; all the other test factors have 2. All combinations of test factors are allowed: Each request contains 1 block with fixed test factor values and the indicated number of hybrid factors with FD values.

Table X gives the array parameters using the hybrid factors. The numbers of hybrid factors ($k^{(2)}$,$k^{(3)}$) are the numbers of pairing and tripling factors corresponding to each example in Table IX; the numbers of associated subarrays of strength 3 and 4 ($n^{(3)}$,$n^{(4)}$) are given for $k = 8$. The blocks column gives the number of FV blocks after all functions have been evaluated. The combinations column shows the numbers of pairwise combinations, as in Table VI. The response time for each request is the difference between its start and stop log entries (fractions of seconds are truncated).

Constraints among the hybrid factors are apparent: The VS (4,5) example has subarrays of test factors with $t' = 4$, so $N \geq 3^1 2^3 = 24$. Without constraints the 2 factors with 6 values would suggest a value for N of at least 36. But these are the pairing factors ($a,$b) and ($b,$c), for which there are only 12 allowed combinations.

B. Examples with Test Factor Constraints

This section presents variable strength results for the shopping cart example. Embedded functions are used for conformance to test model constraints as well as for higher strength subarrays.

The shopping cart example presented in Section II is reused here with hybrid factors to increase the strength of a subarray of 6 test factors: $delChk[0], $newQ[0], $delChk[1], $newQ[1], $delChk[2] and $newQ[2]. Results for strengths $3 \leq t' \leq 6$ are compared. In addition to the earlier model (SCA), which uses parameters intended for a pairwise design, a second model (SCB) uses a parameter adjustment to improve its suitability for higher strengths.

The SCB model reduces the number of item quantity values from 5 (0, 1, 2, 10 and NULL) to 4 (0, 1, 2 and NULL). This is accomplished by removing the value 10 from the functions f_qty, f_newQ_CQ and f_newQ_U, and by removing 10 from the values of $q in the QTY block. This has the effect of reducing the number of values for 1 test factor from 16 to 13, 3 test factors from 5 to 4, 3 test factors from 4 to 3 and 1 test factor from 3 to 2. Consequently SCB response times are reduced considerably.

Table XI shows the variable strength results for shopping cart models SCA and SCB. Strength 2 results without hybrid factors, SCA (2,18) and SCB (2,18) also are shown. The table gives array parameters including the hybrid factors, the numbers of hybrid factors ($k^{(2)}$,$k^{(3)}$) and associated subarrays of strength 3 and 4 ($n^{(3)}$,$n^{(4)}$). The blocks column gives the number of FV blocks after all functions have been evaluated. The combinations column shows the numbers of pairwise combinations, as in the tables above. The response time for each request is the difference between its start and stop log entries.

Each of the SCA and SCB examples used 3 FD blocks, 1 for each of the CHECK, QTY and UPDATE events. Function evaluation for SCA (2,18) yielded 25, 100 and 58 FV blocks respectively for the CHECK, QTY and UPDATE events. Each of the higher strength SCA examples had 9608, 38432 and 2442 FV blocks respectively. Function evaluation for SCB (2,18) yielded 25, 75 and 58 FV blocks respectively for the CHECK, QTY and UPDATE events. Each higher strength SCB example had 4110, 12330 and 974 FV blocks respectively.

Fig. 3 plots the response times on a logarithmic scale. The times for the higher strength SCA examples were from 12 to 60 times longer than those for SCB. However the response time rankings by strength were the same for both models: $t' = 3, 6, 4, 5$, from shortest to longest.

Fig. 4 plots the numbers of test cases generated. For $t' = 3$, 4 and 6, the SCA examples had about twice the number of test cases as the corresponding SCB example; for $t' = 5$, the ratio was 2.7. Again, the rankings by strength were the same for both models: $t' = 3, 4, 6, 5$, from fewest to most test cases.

VI. Discussion

This section interprets the results and relates them to the objectives for this work.

1. *Specify constraints among test factors with simple functions in an established programming language.*

Constraints were described in a language familiar to software engineers to enhance usability and efficiency. Dependent factor values were defined as functions of other, determinant factor values.

The 6 small, strength 2 examples of Table VI, used a total of 9 PHP functions. The longest function had 28 lines; the others had 12 or fewer lines.

The Shopping cart EF request used 9 functions in 3 blocks (11 times fewer than the number of blocks in the FV request) to describe the test factor space more completely. The length of each function was less than 15 lines; 6 of the functions were under 10 lines.

2. *Evaluate composite, embedded functions automatically to generate test cases conforming to test model constraints.*

Automatic evaluation of the functions is important for efficient test design in fast-paced development projects. The complexity of real systems requires composite relations among the embedded functions.

The Calendar, Constraint 1 and Constraint 2 EF examples generated test cases identical to their manually selected FV counterparts. Examination of the test cases for the Constraint 3 and Constraint 4 EF examples indicated conformance to the required constraints. The BMI example showed that strength 2 designs can reach equivalence classes having multiple determinant factors and can pair the classes with their nondeterminant factors.

The Shopping cart EF request used 40 instances of 9 functions in 3 FD blocks. The composite relations among these functions were evident in Table V. The resulting test cases have been made available for examination of their conformance to the constraints defined by the functions.

TABLE X. VARIABLE STRENGTH RESULTS FOR 8 TEST FACTORS

Example (t',k')	Array parameters ($N;t,K,v_1^{k_1}\ldots v_s^{k_s}$)	Hybrid factors ($k^{(2)},k^{(3)}$)	Associated subarrays ($n^{(3)},n^{(4)}$)	Blocks	Combinations Total:Residue	Response time HH:MM:SS
VS (2,8)	$(8;2,8,3^1 2^7)$	(0,0)	(0,0)	1	126:0	00:00:01
VS (2,2)	$(16;2,9,6^1 3^1 2^7)$	(1,0)	(6,0)	6	228:18	00:00:01
VS (3,3)	$(28;2,9,12^1 3^1 2^7)$	(0,1)	(16,5)	12	330:48	00:00:00
VS (4,4)	$(24;2,10,6^1 4^1 3^1 2^7)$	(2,0)	(12,1)	24	320:26	00:00:00
VS (3,5)	$(26;2,12,6^1 4^3 3^1 2^7)$	(4,0)	(22,3)	48	552:66	00:00:01
VS (4,5)	$(29;2,13,6^2 4^3 3^1 2^7)$	(5,0)	(25,5)	48	762:124	00:00:02
VS (5,5)	$(48;2,10,12^1 4^1 3^1 2^7)$	(1,1)	(22,5)	48	446:56	00:00:01
VS (3,6)	$(30;2,14,6^2 4^4 3^1 2^7)$	(6,0)	(32,9)	96	926:140	00:00:02
VS (4,6)	$(36;2,18,6^3 4^7 3^1 2^7)$	(10,0)	(40,15)	96	1856:382	00:00:02
VS (5,6)	$(64;2,12,12^2 8^2 3^1 2^7)$	(0,4)	(44,23)	96	1398:464	00:00:03
VS (6,6)	$(96;2,10,12^1 8^1 3^1 2^7)$	(0,2)	(32,19)	96	562:72	00:00:02

3. Generate variable strength designs using embedded functions.

Section IV introduced hybrid factors to represent allowed combinations of values among 2 or more test factors. The hybrid factors used embedded pairing and tripling functions to increase the strength of selected test factors, the nominal subarray. Methods for obtaining desired strengths of nominal subarrays were specified for strength $t' \leq 6$. Designs generated from these plans in Table IX were summarized in Table X, and their test cases have been made available for review.

Hybrid factors also led to associated subarrays of higher strength: 1 pairing factor combined the pairs of 2 test factors with each of the other k-2 test factors, resulting in k-2 associated subarrays of strength 3. Similarly 1 tripling factor combined the 3-tuples of 3 test factors with each of the other k-3 test factors, resulting in k-3 associated subarrays of strength 4. Table IX listed numbers of associated subarrays of strength 3 and 4 for each of the variable strength examples.

A variable strength design permits a higher strength focus on test factors requiring more attention, i.e. the factors of the nominal subarray. However, the examples of Table IX offer choices with different emphases on the nominal subarray and on the associated subarrays. Thus, a (t',k') = (6,6) design has a higher variable strength t' than a (5,6) design for coverage of the nominal subarray alone. But the (5,6) design has more strength 3 and 4 combinations of the nominal subarray factors with the other test factors. Use of embedded functions enables a variety of variable strength schemes and offers flexibility to define other relations for particular test projects.

4. Generate test case designs with suitable response times.

Different test models typically lead to different sets of test cases. Consequently multiple test case generation runs may be needed to compare alternate test designs. And during the course of a development project, test designs may need to change to reflect system modifications. Thus the usability of a test case generation tool depends on its response times.

The strength 2 results (Table VI) and the small variable strength results (Table X) all had prompt response times; each was less than 10 seconds. The variable strength designs for the SCB test system had response times ranging from 18 to 47 minutes. Overall the response times are acceptable. And there are a number of opportunities to improve response time performance beyond that of this initial implementation.

One element contributing to the response time was the number of FV blocks generated. All blocks had to be processed to complete the design; more blocks required more processing time. For each shopping cart model the number of blocks needed for the intrinsic constraints of the test model was much smaller than the total for the higher strength designs. Table XI showed that for the SCB model the variable strength designs used about 110 times more blocks than the strength 2 design; the corresponding response time ratios exceeded 220.

The increased number of blocks was due to a separate block for each combination of the nominal subarray factor values. This property was apparent in the numbers of blocks for the small examples (Table X).

TABLE XI. VARIABLE STRENGTH RESULTS FOR SHOPPING CART EXAMPLES

Example (t',k')	Array parameters ($N;t,K,v_1^{k_1}\ldots v_s^{k_s}$)	Hybrid factors ($k^{(2)},k^{(3)}$)	Associated subarrays ($n^{(3)},n^{(4)}$)	Blocks	Combinations Total:Residue	Response time HH:MM:SS
SCA (2,18)	$(98;2,18,16^1 5^3 4^6 3^5 2^1 1^2)$	(0,0)	(0,0)	183	2499:383	00:00:09
SCA (3,6)	$(329;2,24,21^1 16^1 13^1 12^1 9^1 8^1 6^1 5^3 4^6 3^5 2^1 1^2)$	(6,0)	(92,9)	50482	9518:2428	03:42:02
SCA (4,6)	$(347;2,28,21^1 20^1 16^1 13^1 12^1 10^1 9^2 8^1 7^1 6^1 5^3 4^6 3^5 2^1 1^2)$	(10,0)	(140,15)	50482	16839:5440	07:28:12
SCA (5,6)	$(1241;2,22,84^1 26^1 24^1 19^1 16^1 5^3 4^6 3^5 2^1 1^2)$	(0,4)	(164,63)	50482	21191:9587	47:23:23
SCA (6,6)	$(656;2,20,37^1 24^1 16^1 5^3 4^6 3^5 2^1 1^2)$	(0,2)	(92,39)	50482	7901:1883	06:16:06
SCB (2,18)	$(55;2,18,13^1 4^6 3^7 2^2 1^2)$	(0,0)	(0,0)	158	1879:307	00:00:05
SCB (3,6)	$(164;2,24,13^2 10^1 9^1 7^1 6^2 4^6 3^7 2^2 1^2)$	(6,0)	(92,9)	17414	6208:1543	00:18:38
SCB (4,6)	$(180;2,28,13^2 12^1 10^1 9^1 8^1 7^3 6^2 4^6 3^7 2^2 1^2)$	(10,0)	(140,15)	17414	10543:3228	00:36:06
SCB (5,6)	$(457;2,22,39^1 20^1 18^1 15^1 13^1 4^6 3^7 2^2 1^2)$	(0,4)	(164,63)	17414	10764:4322	00:47:20
SCB (6,6)	$(319;2,20,22^1 18^1 13^1 4^6 3^7 2^2 1^2)$	(0,2)	(92,39)	17414	4835:1142	00:18:52

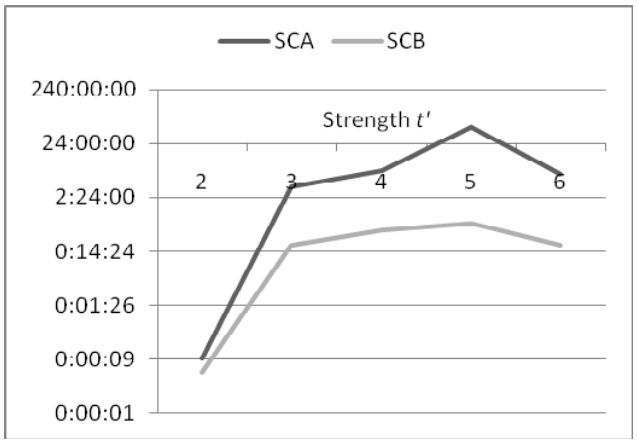

Fig. 3. Response times for variable strength shopping cart examples

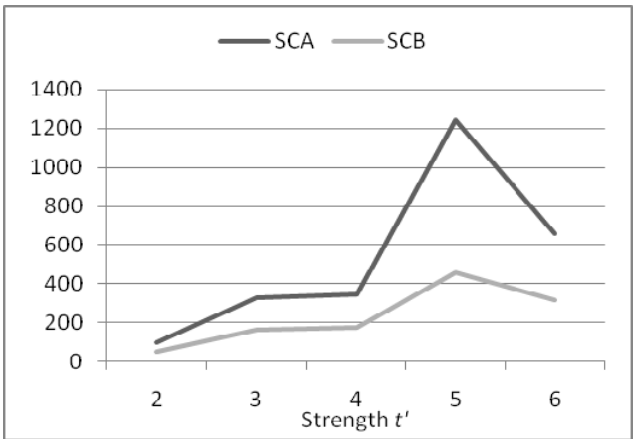

Fig. 4. Numbers of test cases for variable strength shopping cart examples

During evaluation of each hybrid function (a pairing or tripling function), each combination of its determinant factors had a separate block. Evaluation of each subsequent hybrid function led to a separate block for each of its combinations, so the number of resulting blocks was the product of all its combinations with those of the previously evaluated hybrid functions. Thus, after evaluation was complete, there was a separate block for each combination of the nominal subarray factors. However the constant number of blocks could not account for the response time differences among the variable strength designs: The blocks had different factor values.

The variable strength response times did not simply rise monotonically with strength in these examples. It is interesting to note that the response time rankings followed the increasing numbers of associated subarrays (Table XI). The strength 2 design had no associated subarrays and the shortest response time. The $t' = 3$ and 6 designs had the next longer response times. These designs both had 92 associated subarrays of strength 3, and they had 9 and 39 associated subarrays of strength 4 respectively. The $t' = 4$ and 5 designs had the longest response times, and they had 140 and 164 associated subarrays of strength 3 respectively. If this observation holds generally, Table IX would suggest choosing $t' = k'$ or $t' = 3$ for shorter variable strength response times.

5. Generate test case designs with practical sizes.

One of the most important elements of cost in a development project is the number of its test cases. Each case requires time to set up, execute, analyze, and oftentimes rerun. An efficient set of test cases of minimal size is essential.

The strength 2 results (Table VI) showed that the Calendar, Constraint 1 and Constraint 2 EF examples generated the same numbers of test cases as their manually selected FV counterparts. No additional cost in test cases was evident. The Shopping cart EF example generated 98 test cases, while the Shopping cart FV example generated 91. The difference was attributed to a more thorough coverage of the test factor space by the embedded functions test model.

Variable strength models led to additional test cases versus those for strength 2. In the small variable strength results (Table X), the test case ratio for VS (4,4) vs. VS (2,8) was 3. Ratios for $(t',k') = (3,5)$, (4,5), and (5,5) were approximately 3, 4 and 6 respectively. The VS ratios for $k' = 6$ ranged from 4 to 12. For the SCB model (Table XI), test case ratios for SCB (3,6), (4,6), (5,6) and (6,6) were approximately 3, 3, 8 and 6 respectively compared with those for strength 2.

REFERENCES

[1] G. B. Sherwood, "Embedded functions in combinatorial test designs," Proceedings of the 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops, Graz: 2015, pp. 1-10.

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

[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] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, J. S. Collofello, "Variable strength interaction testing of components," Proceedings of the 27th Annual International Computer Software and Applications Conference, Dallas:2003, pp. 413-418.

[6] Testcover.com, LLC. (2016). *Embedded Functions Examples.* Retrieved January 5, 2016, from Testcover.com: http://testcover.com/pub/background/examples2016.php.

[7] M. Achour, F. Betz, A. Dovgal, et al., PHP Manual. Retrieved January 5, 2016, from the PHP Group: http://php.net/manual/en/index.php.

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

[9] Testcover.com, LLC. (2013). *Fixed Values Procedure.* Retrieved January 5, 2016, from Testcover.com: http://testcover.com/pub/fvproc.php.

[10] 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.

[11] Testcover.com, LLC. (2013). *Shopping Cart Example.* Retrieved January 5, 2016, from Testcover.com: http://testcover.com/pub/cartsc.php.

[12] A. W. Williams, R. L. Probert, "A practical strategy for testing pair-wise coverage of network interfaces," Proceedings of the 1996 Seventh International Symposium on Software Reliability Engineering, White Plains, NY: 1996, pp. 246-254.