Equivalence Class Verification and Oracle-free Testing Using Two-layer Covering Arrays

D. Richard Kuhn[1], Raghu N. Kacker[1], Yu Lei[2], Jose Torres-Jimenez[3]

[1] National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{kuhn, raghu.kacker}@nist.gov

[2]Computer Science & Engineering
University of Texas at Arlington
Arlington, TX, USA
ylei@uta.edu

[3]CINVESTAV-Tamaulipas,
Ciudad Victoria, Tamaulipas,
Mexico
jtj@cinvestav.mx

Abstract – **This short paper introduces a method for verifying equivalence classes for module/unit testing. This is achieved using a two-layer covering array, in which some or all values of a primary covering array represent equivalence classes. A second layer covering array of the equivalence class values is computed, and its values substituted for the equivalence class names in the primary array. It is shown that this method can detect certain classes of errors without a conventional test oracle, and an illustrative example is given.**

Keywords-component; combinatorial testing; factor covering array; oracle problem; verification and validation (V&V); t-way testing;

I. INTRODUCTION

In combinatorial testing, as in other approaches, equivalence classes are an essential component. By definition, an equivalence class is a set of variable values where all such values are treated the same by the unit under test. For example, a module that computes shipping cost based on distance d and weight w, may have a few classes for weight, where packages under 1 pound are in one class, 1 to 10 pounds in another, and over 10 in a third class. If the module is specified by some function $f(d,w)$, then $f(d, 0.2) = f(d, 0.9)$, for equal values of d. However, we expect $f(d, 0.2)$ to be different from $f(d, 5.0)$, because two different weight classes are involved.

In general, it should be possible to substitute any value from an equivalence class for any other value from the same class, and leave the result unchanged. If the result changes, then either a) the classes have not been defined correctly for this unit of code; or 2) there is an error in the code. (Note that it is possible that equivalence class values for a particular variable may not be the same in all modules using the variable. For example, another section of the shipping code might have different classes used in deciding what size shipping container to use.)

Thus comparing the result of exercising code with equivalence class values that are expected to produce equal results can be an effective form of verification. If results vary where they should not, then it is possible that equivalence classes have not been defined correctly, and must be fixed before developing unit tests, or a coding error has been discovered. Combinatorial methods can make this process efficient. The process provides a basic check on correctness, detecting a significant class of faults, and it can be fully automated and thus suitable for incorporation into a development environment. Because the testing aspect of this method is based on matching outputs for elements of equivalence classes, we refer to it as *equivalence class value match testing*, or simply *match testing*.

II. METHOD

1. For each variable for which equivalence classes will be established, designate classes and their values as $C_{i,j,k}$, where i indexes variables, j indexes classes, and k indexes values for variable C_i, class j.
2. Compute a primary covering array where factors are variables and levels are variable values for variables without equivalence classes, and equivalence class designations $C_{i,j}$ for variables with equivalence classes.
3. For each row of the primary array, compute a secondary covering array of the factors that are variables with equivalence classes, where levels for each factor $C_{i,j}$ are the values $C_{i,j,k}$ of that class. For each row in the secondary array, substitute its values for the equivalence classes $C_{i,j}$ in the row from the primary array. Thus if the primary array has M rows and each secondary array has N rows, then a full test array with $M \times N$ rows is created. If classes do not all have the same number of values in each, secondary arrays will vary in number of rows, but to simplify the presentation we assume classes with the same number of values.
4. Execute each of the $M \times N$ tests in the test array in blocks of N tests, corresponding to the secondary array generated for each row of the primary array. Verify that output for each of the N tests in a block matches according to a predicate specified for the output.

III. TUTORIAL EXAMPLE

We illustrate the process with a simple example of access control. The rules are that access is allowed if (1) subject is an employee and the time is during working hours and it is a weekday; (2) subject is an employee with administrative privileges; or (3) subject is an auditor and it is a weekday. Based on these rules, equivalence classes can be defined for time of day and day of the week. This is implemented in the code as minutes after midnight for time, with three classes: (0..0539), (0540..1020), (1021..1439). Days of the week can be divided into two equivalence classes, for weekend and weekdays, designated as (1,7) and (2..6) respectively.

The choice of values to be included in equivalence classes may be made using heuristics commonly used in

testing. In many or most cases, boundary values will be effective, but it may also be desirable to include mid-range values if the class represents a continuous-valued variable or ordered set. If the class is an unordered set, values may be selected at random or according to an application-specific heuristic, such as the operational profile of the application. With an unordered set, match testing may be less effective in catching errors, as part of its effectiveness stems from the manner in which incorrect relational operators produce variations in results for some tests of the secondary array.

For readability in this simple example, we have designated different results for different segments of the code as return values of 1, 2, and 3. As noted above, in reality results would be defined and differentiated by some predicate, not necessarily a single possible value. For instance, in our access control example below, decision 3 might be recognized by system effects such as the appearance of an auditor role in the system log file. It is also possible that ranges of values for more than one variable may be specified in a particular output predicate.

We illustrate the method using the small program in Figure 1. Faults are introduced into various versions by mutating relational operators in function `access_chk()`.

```
#include <stdio.h>
static int START = 0540;
static int END = 1020;
static int MON = 2;
static int FRI = 6;
int emp;  // employee
int d;    // day, 1..7
int t;    // time, minutes
int p;    // priv
int aud;  // auditor

main(argc, argv)
int argc;
char *argv[];
{
    if(argc < 6) {fprintf(stdout, "Error: Command
    line arguments are\n"); exit(1); }
    emp = atoi(argv[1]);
    d = atoi(argv[2]);
    t = atoi(argv[3]);
    p = atoi(argv[4]);
    aud = atoi(argv[5]);
    fprintf(stdout, "%d\n", access_chk());
}

int access_chk() {
   if (emp && t >= START && t <= END &&
   d >= MON && d <= FRI) return 1;
   else
   if (emp && p) return 2;
   else
   if (aud && d >= MON && d <= FRI)
   return 3;
   else
   return 0;
}
```

Figure 1. Example program under test.

Test Generation

The primary array includes factors for the two equivalence classes and other variables.

```
emp:  boolean
day:  (1,7), (2,6)  -> classes A1, A2
time: (0,0539), (0540,1020), (1021, 1439)
      -> classes B1, B2, B3
priv: boolean
aud:  boolean
```

Factors and levels used to generate the primary covering array are thus:

```
emp (bool) : 0,1
day (enum) : A1,A2
time (enum): B1,B2,B3
priv (bool): 0,1
aud (bool) : 0,1
```

Pairwise coverage is obtained with the following array:

```
0,A2,B1,1,1
1,A1,B1,0,0
0,A1,B2,1,0
1,A2,B2,0,1
0,A1,B3,0,1
1,A2,B3,1,0
```

Secondary arrays are computed to implement pairwise tests for each row of the primary array. Thus the first row generates:

```
0 2 0 1 1
0 6 0 1 1
0 2 539 1 1
0 6 539 1 1
```

Test Results

If equivalence classes have been defined correctly, and there are no errors in the code, then results should be the same (as defined by some predicate) for each set of tests in the secondary array generated from one row of the primary array. Thus for the correct code, results are as follows.

```
3333
0000
0000
1111
0000
2222
```

Each row corresponds to one row of the primary covering array, and each column gives the result for one of the secondary array tests generated for the corresponding primary array row. Note that all values are identical for columns of a given row. Because the equivalence classes have been defined correctly and the code is correct, equivalent values produce the same results.

With the mutation below, where `t <= END` has been replaced with `t == END`,

```
if (emp && t>=START && t==END
&& d>=MON && d<=FRI) return 1;
```

the result is as follows. Note that values differ in the fourth row, because the elements of the equivalence classes for

time of day no longer produce the same result. Thus we have detected an error in the code.

```
3333
0000
0000
3311
0000
2222
```

A set of 10 mutated programs was generated, with the fault detection results as shown in Table I.

TABLE I. FAULT DETECTION RATES FOR SEEDED ERRORS

Version	Mutated Code	Fault detected
1	(emp && **t>START** && t<=END && d>=MON && d<=FRI)	YES
2	(emp && t>=START && **t==END** && d>=MON && d<=FRI)	YES
3	(emp && t>=START && t<=END && d>=MON && **d<FRI**)	YES
4	(emp && t>=START && t<=END && **d>MON** && d<=FRI)	YES
5	(aud && d >= MON && **d<FRI**)	YES
6	(emp && t>=START || t<=END && d>=MON && d<=FRI)	NO
7	(emp && t>=START && t<=END || d>=MON && d<=FRI)	NO
8	(emp && t>=START && t<=END || d>=MON || d<=FRI)	YES
9	(aud && d >= MON || d <= FRI)	YES
10	(aud && **d <= MON** || d <= FRI)	YES

IV. REALISTIC EXAMPLE

To illustrate the application of this method as it can be applied in practical testing, we use a Traffic Collision Avoidance System module [6], which has been included in many studies of test methods. Although small, the TCAS module code is a realistic example for match testing, which as noted previously is intended for module or small unit testing. The code includes a set of 41 versions with seeded faults. Roughly two thirds of the faults are simple changes such as replacing a constant with another constant, replacing >= with >, or dropping a condition. The TCAS program has 12 input variables specifying parameters of two aircraft, such as speed and position, and one output variable. For testing studies, tests are run against the set of faulty versions to determine which can be detected by the test set.

For this example, we developed equivalence classes for three of the variables and produced two separate two-layer test covering arrays in 3-way×3-way and 4-way×3-way configurations. The number of tests and results are shown in Table II.

TABLE II. TCAS TESTS AND RESULTS.

Primary x secondary	#tests	total	faults detected
3-way x 3-way	285x8	2280	6
4-way x 3-way	970x8	7760	22

Although a large set of tests is required, the number is practical for most applications because no test oracle is needed. Once equivalence classes have been defined, tests can be run in parallel if desired. Results are encouraging, as more than half of the 41 faults were detected with the second configuration. Because match testing can be fully automated, these faults could be detected without human effort required to develop test oracles.

V. DISCUSSION

A significant class of faults can be detected with this method, which can be automated and implemented without the need for conventional test oracles. Using both primary and secondary covering arrays makes it possible to find faults that might not be discovered with a simpler implementation of the equivalence class verification. Table III shows the number of faults detected using comparisons of results for various combinations of the equivalence class values for the example in Figure 1. In the table, "L" refers to the lower value of an equivalence class and "H" refers to the higher value. Thus "LL/HH" indicates comparing results for the lower values of day and time variables with results for higher values of these. The number of faults detected varies, ranging from 2 to 8, and only two of the value selections detect eight faults.

TABLE III. FAULT DETECTION WITHOUT SECONDARY ARRAY.

faulty version	LL/ HH	HL/ HH	LH/ HH	LL/ HL	LL/ LH	HL/ LH
1	Y	Y	N	N	Y	Y
2	Y	Y	N	N	Y	Y
3	Y	N	Y	Y	N	Y
4	Y	N	Y	Y	N	Y
5	Y	N	Y	Y	N	Y
6	N	N	N	N	N	N
7	N	N	N	N	N	N
8	Y	N	Y	Y	N	Y
9	Y	N	Y	Y	N	Y
10	Y	N	Y	Y	N	Y

The example in Figure 1 also illustrates the limitations of this method. Note that faults in versions 6 and 7 are not detected. Detecting either of these mutations requires the faulty expression to evaluate to a different truth value than the correct version, but no set of values from the same equivalence classes will produce this result. In some cases, such problems can be resolved by defining a more comprehensive input model. The completeness of the input model, in turn, can be validated by ensuring that tests derived from it produce a sufficiently high level of structural coverage (e.g., statement or branch coverage).

The method is not limited to simple predicates as included in the example, and can be effective in any case where the faulty predicate maps elements of a single equivalence class to two or more different results. Note that source code is not required, provided specifications are sufficiently detailed to allow identification of equivalence

classes. As such, the method may be applied to system testing in some cases. We anticipate its primary utility in unit testing, because small units are likely to have more tightly defined equivalence classes, but this is a practical rather than theoretical limitation.

The number of tests required by the two-layer arrays used in match testing will be generally much larger than what would be produced if the union of equivalence class values for each variable in the secondary array are treated as values for a single variable in a covering array. However, the time and cost will be essentially insignificant for the two-layer array because an oracle is not needed. Conventional testing using a single covering array would require producing a test oracle, typically a very expensive process. For example, using a single covering array for the example in Sect. IV requires 1177 3-way tests and 4816 4-way tests, significantly larger than the test sets shown in Table II. But the cost of producing test oracles for several thousand tests is likely to be very large, whereas the fault detection shown in Table II has virtually no cost because it is fully automatic.

One complication is that equivalence classes may be defined by more than one relation. Returning to the example in the introduction, the specification may include different processing for cases specified by multiple conditions. The constraint handling features of ACTS [1][2] or other covering array generators may be applied to produce equivalence classes for the primary array that are then expanded in the secondary array. It is also possible that equivalence classes may be defined by more than one variable, a circumstance where constraints may be used in some cases, or where tests may be eliminated (possibly at the cost of reduced fault detection).

It is important to keep in mind that the match testing method is not intended as a replacement for conventional testing. As can be seen in the examples, in most cases it will not detect all faults. Rather, this method can serve as an inexpensive pre-testing phase that can detect a significant proportion of errors and possibly reduce overall test cost.

VI. Related Work

Few methods exist for testing without conventional test oracles. One recent approach is metamorphic testing [3], which uses one or more metamorphic relations defining properties relating test inputs and outputs, with subsequent transformations of test data that can then be checked for conformance to the metamorphic relations. For example, since $cos(x) = cos(x+360)$, test output of a cosine function for x would be compared with test output of the function for $x+360$, with a difference indicating an error. Partial oracles are another approach somewhat different from conventional testing, in which properties of output are checked, rather than expecting a specific output value for a given input. A third approach is the established practice of including assertions in code, to ensure that various properties are maintained during execution.

While not using a test oracle in the conventional sense, with a particular output expected for a given input, all of these methods rely on some specification relating test inputs to outputs. Match testing does also, in that we use information that is latent in the specification of equivalence classes. Metamorphic testing, partial oracles, and assertion checking, in contrast, use specification information that relates inputs to outputs directly using some property. Match testing does not conflict with these approaches, and might be used to improve their efficiency.

VII. Conclusions

Match testing is designed to be incorporated into a test development environment, and can be fully automated. Test designers may define equivalence classes and have these verified by executing the code and comparing results of each class. While not always suitable for large modules, the method provides a basic check on the soundness of equivalence classes, while detecting faults that may escape detection with conventional testing. Future work will integrate the method into a test development environment such as ComTest [4], integrated with CITLab [3].

A large-scale evaluation of the method is planned, applying it to a set of realistic sized programs, and characterizing the test set size and cost in comparison with conventional combinatorial testing. We also plan to investigate special considerations for floating point variables, and the possibility of generating a covering array using a sampling of each equivalence class involved. Another research question is how this method can be used most effectively in the testing process.

Acknowledgment: J. Torres-Jimenez's work was partially funded by the following projects: 51623 - Fondo Mixto CONACyT y Gobierno del Estado de Tamaulipas; 238469 – CONACyT Métodos Exactos para Construir Covering Arrays Óptimos; 232987 - CONACyT Conjuntos de Prueba Óptimos para Métodos Combinatorios.

Disclaimer: *Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.*

References

[1] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: A general strategy for t-way software testing. *14th international conference on the engineering of computer-based systems*, 2007, pp 549–556

[2] ACTS Home Page, http:// csrc.nist.gov/acts/

[3] T.Y. Chen, T.H. Tse, and Z.Q. Zhou, "Fault-based testing without the need of oracles", *Information and Software Technology*, vol. 45, no. 1, pp. 1–9, (2003).

[4] https://github.com/comtest/comtestnist/releases

[5] A. Gargantini, P. Vavassori, "CITLab, a Laboratory for Combinatorial Interaction Testing", *IWCT* 2012.

[6] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow-and control flow-based test adequacy criteria. *Proc. Sixteenth Int. Conf. Software Engineering*, pp. 191–200, May 1994