Edgar H. Sibley
Panel Editor

Orthogonal Latin squares—a new method for testing compilers—yields the informational equivalent of exhaustive testing at a fraction of the cost. The method has been used successfully in designing some of the tests in the Ada Compiler Validation Capability (ACVC) test suite.

ORTHOGONAL LATIN SQUARES: AN APPLICATION OF EXPERIMENT DESIGN TO COMPILER TESTING

ROBERT MANDL

When writing test suites for testing a compiler or any other program or system, it is often necessary to test a set of states—statements, complex constructs, data paths, etc.—whose form or action depends on a number of factors, each of which spans an associated range. If the factors are independent, the state space is the Cartesian product of the ranges, which reflects the orthogonality of the system. For example, in simple declarations like

```
'static int ABCDE;'
```

in C, `'static'` can be replaced by any member of the set {`'auto'`, `'extern'`, `'register'`, `'static'`}, and `'int'` by any member of the set {`'int'`, `'float'`, `'char'`, `'unsigned int'`}. This results in a legal declaration for which we can then check correct allocation of storage, correct initialization, etc. In this perhaps overly simplified example, the state space consists of all possible selections of two items: one from the list of storage-class specifiers (first list), and one from the list of type specifiers (second list) for a total of 16 selections. If we want to test the operation of binary arithmetic operators working on such values, we add a third list, {`'+'`, `'-'`, `'/'`, `'*'`}. The size of the resulting state space now becomes 64, because there are 4^3 ways of selecting one item per list from the three four-element lists. The original example `'static int ABCDE;'` is orthogonal because the two selection processes can take place independently of each other and in any order. Since the designers of C have decreed that taking the ratio of two values of type `'char'` and taking the product of two values of type `'char'` are meaningful operations, the expanded example is also orthogonal. However, not everything in C is orthogonal. For instance, it is legal to define functions returning numeric types and functions returning pointers, but not functions returning arrays or structures; one can define arrays of just about anything, but not arrays of functions.

To illustrate the use of the orthogonal-Latin-squares method, we present an example derived from an actual ACVC[1] test that utilizes the method. The test verifies that, in Ada®, ordering operators on enumeration values are evaluated correctly even when these enumeration values are ASCII characters.

THE EXAMPLE: ADA COMPILER VALIDATION

In Ada, when a programmer defines an enumeration type, the type definition lists all the enumeration liter-

This work was performed at Softech (Waltham, Massachusetts) under contract MDA-903-79-C0687.

© 1985 ACM 0001-0782/85/1000-1054 75¢

[1] Ada Compiler Validation Capability.
Ada is a trademark of the U.S. DoD (Ada Joint Program Office).

als that are to constitute the values of the type being defined, for example,

```
TYPE thermostat_belief IS
  (too_cold_in_here, too_hot_in_here,
   just_right);
TYPE gyrostabilizer_belief IS
  (plus, minus, just_right);
TYPE openers IS ('{', '(', '[');
```

If an enumeration literal appears in at least two such definitions ("is overloaded"), qualifying the literal by the name of the type to which it belongs will remove the ambiguity:

```
IF thermostat_belief'(just_right)
  = ... THEN ...;
```

There are two kinds of enumeration literals: user-selected identifiers and character literals. ASCII characters automatically belong to the built-in type CHARACTER, for which the ordering relationship is the ordinary ASCII collating sequence. Therefore, ASCII characters that have been explicitly listed in user-defined enumeration types are overloaded. (It is the programmer's responsibility to avoid situations where the overloading cannot be resolved by the compiler.)

One particular complication of Ada's enumerated types is that (although characters already have an implied ordering, which for letters amounts to the alphabetic order), when the ordering operator is applied to two literals (more precisely, to two instances of literals in the program text) that are known to be functioning as enumeration literals and happen also to be ASCII characters, it is the enumeration order that must be upheld, not the predefined default ordering for ASCII characters. In other words, the test objective is *Assume that a user-defined enumeration type imposes on some of the letters of the alphabet an ordering that is different from the standard lexicographical ordering. Check that, when the ordering operators are applied to such letters (qualified as members of the user-defined enumeration type), the answers produced reflect the new definition and not the standard lexicographical ordering.* To implement this objective, an enumerated type must be defined that is not consistent with the natural ordering of the characters and in which some of the ordering relationships between individual characters are the same as in ASCII while others are different. Enumerated types may contain mixtures of character literals and user-named identifiers; we assume that the test in question is restricted to enumerations consisting exclusively of ASCII characters.

The first step is to test the correctness of the operators on pairs that contain an element that is first or last in the enumeration, and also on pairs that do not contain such extreme values. In order not to expand the size of state space unnecessarily, we settle on four values in the enumerated type—the four letters (characters) S, P, M, and R, in this order:

```
TYPE my_enumeration_type IS
  ('S', 'P', 'M', 'R');.
```

In addition to the four literals, we also introduce four variables of this type and initialize them to the same four values. (We have replaced the name 'my_enumeration_type' by the shorter name 't'.)

```
mvar : t := t'('M');
pvar : t := t'('P');
rvar : t := t'('R');
svar : t := t'('S');
```

As it may make a difference whether a value is used as a left operand as opposed to a right operand, we consider the two operands as separate relevant features, as separate factors in the generation of state space. Two other determining factors are the identity of the operator and the pattern of distribution of literal values among the operands.

In our example, there are four operators in the "ordering-operator" class. Although we would have preferred to extend the scope to cover equality/inequality operators in addition to the ordering operators, for reasons that will soon become apparent we had to restrict ourselves to the four ordering operators. Since we are not concerned here with expression evaluation, we distinguish only two kinds of expression operands: literals—specifically, character literals considered as elements of a user-defined enumerated type—and variables of the user-defined enumerated type. Both operands may be literals, or neither of them, or precisely one (the left or the right); thus, this factor also has four distinguishable levels.

In short, we have isolated the following relevant factors, each of which has four levels:

the value of the left operand
  ('S', 'P', 'M', 'R'),
the value of the right operand
  ('S', 'P', 'M', 'R'),
which operator is used
  ('<', '<=', '>', '>='), and
the literality pattern for the two operands:

```
literal    vs. literal,    as in t'('P') < t'('R')
literal    vs. variable,   as in t'('P') <    rvar
variable   vs. literal,    as in   pvar  < t'(rvar)
variable   vs. variable,   as in   pvar  <    rvar.
```

This makes a total of 256 test cases (n^k combinations in the case of k factors with n levels each), which take the form

```
IF t'('S') >= t'(svar) THEN ...; END IF;.
```

In most situations, testing all these combinations would be prohibitively expensive. We argue in this article that in many cases testing *every single* combination is not really necessary. Assume, for the sake of argument, that we really intended to test all 256 combinations, but somehow managed to perform only 255 or 254. We might argue that 254 are almost as good; and that if all 254 have passed, then surely all relevantly distinct combinations must have been tested and the missing cases would surely have passed also. (Of course, if only one of the 254 tests produced a wrong result, this would be absolute proof of failure, indepen-

dently of anything that the remaining tests could present.) The method proposed here attempts to formalize the phrase "surely all relevantly distinct combinations."

THE PROBLEM

Given a test objective (not necessarily in the context of compiler testing), we identify a state space spanned by a finite number of variables with a number of allowable values (or representative values, if the test objective allows it). Two traditional approaches are (1) exhaustively covering the state space, and (2) making a random selection of test cases from the state space. Although exhausting the state space may be ideal, the state space may be prohibitively large. The other approach—making a random selection—may of course uncover deficiencies; but even if it does, it is difficult to assess the level of confidence to be derived from the apparently successful testing.

If we accept the premise that in some situations it is possible to achieve a high level of confidence by performing a suitably chosen nonexhaustive test procedure, we must make explicit what conditions would render such a procedure satisfactory—perhaps even as satisfactory as the exhaustive testing.

Suppose that all potential combinations are divided into a reasonable number of distinct classes by applying some criterion that places two combinations in the same class if and only if the testing of one combination provides nothing new over what is available from the testing of the other and combinations from the other classes. In this case, testing one representative combination from each such class would be perfectly satisfactory. Actually, it is unnecessary to define classes such that *any* selection of one representative per class would provide as much confidence as exhaustive testing; it is sufficient to provide *one* selection rule that provides that level of confidence—and this is what we are attempting to do.

RELEVANCE OF LATIN SQUARES

In the ACVC example, there are two factors (known as *index factors*) whose levels reflect the values of the two operands, as well as a number of factors reflecting the shape of the construct (*contents factors*). There are two contents factors: One reflects the identity of the operator, and the other the *literality pattern* for its operands. When the matrix representation of the test plan is presented later in terms of matrices that are Latin squares, the index factors will correspond to the indexes of the matrix, and the contents factors will correspond to the contents of the matrix entries. Since matrices are two-dimensional arrays, the number of index factors is always 2.

In a faulty implementation, it is quite possible that some operator will always return a value opposite to the value expected, or will always return TRUE, or that one or the other of these situations will obtain if, and only if, a specific form of the construct is used. However, it is less likely that two operands that both evaluate to 17 will be recognized as equal while two operands that evaluate to 18 will not. Therefore, although we do require that every combination be "represented", we do not insist that every combination be present with *all* possible operand values. What we do require is that for any given ordered pair of contents-factor levels (not both belonging to the same factor) *there exist* an ordered pair of index-factor levels such that the given pair of contents-factor levels is tested in the context provided by the pair of index-factor levels. We then know that every single combination of contents-factor levels has been tested in some context (although not necessarily with all possible combinations of value parameters). More precisely, we know that every combination of contents-factor levels has been tested in exactly one context (with exactly one combination of value parameters).[2]

The design of test cases is similar in some respects to the design of statistical experiments. From the variety of experiment-design methods used in statistics, the most useful to us seems to be Latin squares. A Latin square is a balanced two-way classification scheme that is usually represented as a square matrix. For example, the matrix

A	B	C	D
C	D	A	B
D	C	B	A
B	A	D	C

reflects a scheme with a total of three variables, each having four levels. It might be used to test the application of four different treatments A, B, C, D (variable 3) to four different crops (rows 1, 2, 3, 4) (variable 2) using, one at a time, four different application methods (columns 1, 2, 3, 4) (variable 1). In this scheme, each of the four treatment levels A, B, C, D appears precisely once in each column of the matrix; and each of the four treatment levels A, B, C, D appears precisely once in each row of the matrix. With the rows and the columns labeled, the matrix appears as follows:

	Application method			
	#1	#2	#3	#4
Crop #1	A	B	C	D
Crop #2	C	D	A	B
Crop #3	D	C	B	A
Crop #4	B	A	D	C.

Testing all possible combinations in the above example would require 64 tests (= 4^3), whereas the Latin-squares method will yield much the same information at the cost of performing only 16 tests.

The four treatments of the respective crops could have been four different dosages, or levels, of one and the same substance. The generalization from "4 crops, 4 methods, and 4 treatment levels" to "n crops, n methods, and n treatment levels" is immediate. Exhaustive testing in this situation would require n^3 tests, whereas Latin-squares testing requires only n^2 tests.

To investigate the impact of the presence of another substance—a fourth variable—(also in four levels—

[2] This methodology can be generalized in several ways: by increasing the number of contents factors; and by using *balanced incomplete block designs* [5], where the combinations, instead of each occurring exactly once, occur exactly twice, or exactly λ times for some λ.

alpha, beta, gamma, delta), one could perform two separate experiments, each described by a Latin square as above; however, this scheme would yield no information about any effect of the simultaneous presence of the two substances—the substance represented by the Latin letter and the substance represented by the Greek letter. An economical design for studying the combined effects would consist of a Greco–Latin square

A alpha	B beta	C gamma	D delta
C delta	D gamma	A beta	B alpha
D beta	C alpha	B delta	A gamma
B gamma	A delta	D alpha	C beta

where each of the 16 combinations of a Latin letter with a Greek letter occurs precisely once. A Greco–Latin square results from combining, entry by entry, two Latin squares that are orthogonal to each other in the sense that they satisfy the following *orthogonality* condition:

Two Latin squares are orthogonal if, when they are combined entry by entry, each pair of symbols occurs precisely once in the combined square.

The number of index factors is always two because they correspond to the indexes (dimensions) of a two-dimensional array. If we have identified k contents factors (with n levels each), then we need a set of k mutually orthogonal $n \times n$ matrices (k can be 1, 2, 3, or more). Thus, we have reduced the problem of finding suitable designs to that of finding sets of orthogonal Latin squares.

Some basic properties of orthogonal Latin squares are given in Figure 1. Further good examples of Greco–Latin and orthogonal Latin squares can be found in [4, Tables 15 and 16].

1. For $n = 3, 4, 5, 7, 8$, and 9, one can find $n - 1$ distinct, mutually orthogonal $n \times n$ Latin squares.
2. The largest possible number of orthogonal $n \times n$ Latin squares is $n - 1$.
3. If the factorization of n into powers of distinct primes $p_1, p_2, \ldots, p_r$ $(p_1 < p_2 < p_3 < \cdots)$ is

$$p_1^{q_1} \times p_2^{q_2} \times \cdots p_r^{q_r},$$

then there exist at least

$$p_1^{q_1} - 1$$

orthogonal Latin squares of order n [6].
(This theorem guarantees three orthogonal Latin squares if n is a multiple of 4, and nothing at all if n is an even integer not divisible by 4. Euler conjectured [6] that for such "unevenly even numbers" orthogonal Latin squares do not exist. The conjecture was disproved in 1959 [1, 2, 7].)
4. For $n = 6$, orthogonal Latin squares do not exist.*
5. For every integer n greater than 6, there exists a pair of orthogonal Latin squares of order n. (The existence of a pair of orthogonal Latin squares of some order does not imply that for any Latin square of that order one can find a Latin square orthogonal to it.)

* This is why we cannot deal with six-level variables.

FIGURE 1. Facts about Pairs or Sets of Orthogonal Latin Squares

ACHIEVING NEAR-EXHAUSTIVE TESTING THROUGH EXPERIMENT DESIGN

Returning now to the problem of devising a suitable set of test cases for near-exhaustive testing (whether for compiler testing or testing in a more general setting), we propose a method borrowed from statistical experiment design that utilizes the properties of orthogonal Latin squares and balanced incomplete block designs.

For k variables each admitting n values (that is, for which we recognize n levels as being relevantly distinct; RESTRICTION: $k <= n + 1$), choose a set of $k - 2$ orthogonal $n \times n$ Latin squares and implement, instead of the total number n^k of test cases, only the n^2 combinations corresponding to the entries of the square. This guarantees the "essential exhaustiveness" at a substantially lower cost (e.g., in the case of $n = 4$ and $k = 4$, 16 test cases instead of 256). Of course, a single Latin square is sufficient in the case of three variables; and a Greco–Latin square in the case of four variables. As can be seen from Figure 1 showing the properties of orthogonal Latin squares, Greco–Latin squares exist for all values of n other than 6. Although large sets of mutually orthogonal Latin squares do exist, it is usually not necessary to resort to them, since most practical situations do not involve more than four variables.

Occasionally, one may want to choose a Greco–Latin square having additional properties. For instance, if the diagonal of the square has some special significance in the problem space, one might want to choose a Greco–Latin square in which the Latin diagonal and the Greek diagonal each contain n distinct values. Furthermore, since the diagonal $\langle 1, 1\rangle, \langle 2, 2\rangle, \ldots, \langle n, n\rangle$ is unsatisfactory in some cases, one could choose a square whose diagonals are nontrivial permutations of the full set of n values. This was done in the test that served as the basis for the example discussed in this paper: The Greco–Latin square representing this test is given in Figure 2, and the actual code for the relevant

Left operand	Right operand			
	S	P	M	R
S	A-ALPHA	B-BETA	C-GAMMA	D-DELTA
P	C-DELTA	D-GAMMA	A-BETA	B-ALPHA
M	D-BETA	C-ALPHA	B-DELTA	A-GAMMA
R	B-GAMMA	A-DELTA	D-ALPHA	C-BETA

```
(       A = '<'     ALPHA = VV
        B = '<='    BETA  = VL
        C = '>'     GAMMA = LV
        D = '>='    DELTA = LL    )
```

FIGURE 2. The Greco-Latin Square Used in the Example

```
IF t'(svar) <  t'(svar)  THEN  bump ;                  END IF;
IF t'(svar) <= t'('P' )  THEN  NULL;   ELSE  bump ;   END IF;
IF t'('S' ) >  t'(mvar)  THEN  bump ;                  END IF;
IF t'('S' ) >= t'('R' )  THEN  bump ;                  END IF;

IF t'('P' ) >  t'('S' )  THEN  NULL;   ELSE  bump ;   END IF;
IF t'('P' ) >= t'(pvar)  THEN  NULL;   ELSE  bump ;   END IF;
IF t'(pvar) <  t'('M' )  THEN  NULL;   ELSE  bump ;   END IF;
IF t'(pvar) <= t'(rvar)  THEN  NULL;   ELSE  bump ;   END IF;

IF t'(mvar) >= t'('S' )  THEN  NULL;   ELSE  bump ;   END IF;
IF t'(mvar) >  t'(pvar)  THEN  NULL;   ELSE  bump ;   END IF;
IF t'('M' ) <= t'('M' )  THEN  NULL;   ELSE  bump ;   END IF;
IF t'('M' ) <  t'(rvar)  THEN  NULL;   ELSE  bump ;   END IF;

IF t'('R' ) >  t'(svar)  THEN  bump ;                  END IF;
IF t'('R' ) >= t'('P' )  THEN  bump ;                  END IF;
IF t'(rvar) <  t'(mvar)  THEN  NULL;   ELSE  bump ;   END IF;
IF t'(rvar) <= t'('R' )  THEN  bump ;                  END IF;
```

Note—*bump* is a procedure that bumps an error-count variable.

FIGURE 3. The Code for the Relevant Portion of the ACVC Test

portion of the test is presented in Figure 3.

The orthogonal Latin squares method described here has been successfully used in a number of tests developed at Softech for use in the ACVC test suite. The examples are based on tests C45210A.ADA of October 15, 1980, and C83A05A.ADA of February 11, 1980. Both tests, as well as several others, include extensive commentary describing the method in general and the method as applied specifically to these tests. Most of these comments (unless repetitive) have been incorporated in some form in this paper.

CONCLUSIONS

An exhaustive test suite for a state space with k n-valued variables would enumerate all the ordered k-tuples where each position has n possible values, at a cost of n^k (the number of tests required), which is sometimes more than we can afford.

A reasonable random selection of test cases would probably be about the same size as a test suite that simply takes the k variables in turn, and includes for each of them n tests to cover the n legal values of the variable; its cost in terms of tests would also be about the same, $n \times k$, compared to n^2 for the orthogonal-squares method. However, the random-selection approach is much less effective in detecting malfunctions and yields much less information when it does in fact detect something.

The orthogonal-Latin-squares method being proposed seems to strike a very efficient compromise between the level of effort required and the amount of information obtained: At a cost commensurate with that of a traditional set of test cases selected at random, it yields about as much useful information as the prohibitively expensive exhaustive testing.

Acknowledgments. The author would like to thank John R. Kelly, of Softech, for a critical reading of an early version of this paper.

REFERENCES

Note: References [3] and [8] are not mentioned in the text.

1. Bose, R.C., and Shrikhande, S.S. On the construction of sets of mutually orthogonal Latin squares and the falsity of a conjecture of Euler. *Trans. Am. Math. Soc. 95* (1960), 191–209.
2. Bose, R.C., Parker, E.T., and Shrikhande, S.S. Further results on the construction of mutually orthogonal Latin squares and the falsity of Euler's conjecture. *Can. J. Math. 12* (June 1960), 189–203.
3. Diamond, W.J. *Practical Experiment Designs for Engineers and Scientists.* Lifetime Learning Publications, Belmont, Calif., 1981.
4. Fischer, R.A., and Yates, F. *Statistical Tables for Biological, Agricultural, and Medical Research.* 2nd ed. Oliver and Boyd, London, 1943.
5. Hall, M., Jr. *Combinatorial Theory.* Blaisdell Publishing Co., Waltham, Mass., 1967.
6. MacNeish, H.F. Euler squares. *Ann. Math. 23*, 3 (Mar. 1922), 221–227.
7. Parker, E. Construction of some sets of mutually orthogonal Latin squares. *Proc. Am. Math. Soc. 10*, 6 (Dec. 1959), 946–949.
8. Winer, B.J. *Statistical Principles in Experimental Design.* 2nd ed. McGraw-Hill, New York, 1971.

CR Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*validation*; G.1.0 [**Numerical Analysis**]: General—*error analysis*; G.3 [**Probability and Statistics**]

General Terms: Verification

Additional Key Words and Phrases: Greco-Latin squares, orthogonal Latin squares

Received 2/83; revised 12/84; accepted 3/85

Author's Present Address: Robert Mandl, Analogic Corporation, Audubon Road, Wakefield, MA 01880.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.