Measurement 46 (2013) 3745–3752

Contents lists available at SciVerse ScienceDirect

Measurement

journal homepage: www.elsevier.com/locate/measurement

Combinatorial testing for software: An adaptation of design of experiments

Raghu N. Kacker [a,*], D. Richard Kuhn [a], Yu Lei [b], James F. Lawrence [a,c]

[a] National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
[b] University of Texas, Arlington, TX 76019, USA
[c] George Mason University, Fairfax, VA 22030, USA

ARTICLE INFO

Article history:
Available online 14 March 2013

Keywords:
Computer security
Covering arrays
Design of experiments
Measurement
Metrology
Orthogonal arrays
Software engineering
Software testing

ABSTRACT

Software has become increasingly ubiquitous in tools and methods used for science, engineering, medicine, commerce, and human interactions. Extensive testing is required to assure that software works correctly. Combinatorial testing is a versatile methodology which is useful in a broad range of situations to detect faults in software. It is based on the insight that while the behavior of a software system may be affected by a large number of factors, only a few factors are involved in a failure-inducing fault. We discuss the development of combinatorial testing for software as adaptation of design of experiment methods. Combinatorial testing began as pairwise testing in which first orthogonal arrays and then covering arrays were used to make sure that all pairs of the test settings were tested. Subsequent investigations of actual software failures showed that pairwise (2-way) testing may not always be sufficient and combinatorial *t*-way testing for *t* greater than 2 may be needed. Until recently efficient tools for generating test suites for combinatorial *t*-way testing were not widely available. Combinatorial testing has become practical because efficient and free downloadable tools with support of constraints have become available.

Published by Elsevier Ltd.

1. Introduction

A software fault is a mistake in the code which when encountered may cause the software to fail. Failure means that the software behaves in unexpected (incorrect) ways. A variety of testing methods are used to avoid, detect, and correct faults during and after development of software. A 2002 report estimated that the cost to the US of inadequate infrastructure for software testing was between \$22.2 and \$59.5 billion [1]. In a decade since then software has become more complex and the detection of faults has become more challenging. An often used approach for detecting software faults is dynamic testing in which the software system under test (SUT) is exercised (run) for a set of test cases, the expected (correct) behavior of the system is predetermined for each test case, and the actual behavior is compared against the expected. The SUT passes a test case when the behavior is as expected and fails when the behavior is different from the expected behavior. When the SUT fails for one or more test cases the underlying faults in the software which induce the failure are searched and then corrected. The SUT could be any part of a software system however small or large for which test cases can be constructed, the expected (correct) behavior can be determined for each test case, tests executed, and the actual behavior can be observed and assessed. Dynamic testing is often used for independent verification and validation of software systems.

Combinatorial testing is a type of dynamic testing in which distinct (but possibly related) test factors are specified from the requirements, knowledge of system implementation and internal operations, and other information available about the SUT. The possible values of test factors may reside on continuous or discrete scales. In either case, for each test factor relatively few discrete test settings are

* Corresponding author. Tel.: +1 301 975 2109.
E-mail address: raghu.kacker@nist.gov (R.N. Kacker).

0263-2241/$ - see front matter Published by Elsevier Ltd.
http://dx.doi.org/10.1016/j.measurement.2013.02.021

specified by equivalence partitioning, boundary value analysis, and expert judgment. Then each test case is expressed as a combination of one test setting for every test factor [2–5]. Suppose the first test factor has v_1 test settings, the second test factor has v_2 test settings, etc., and the k-th test factor has v_k test settings, where $v_1, v_2, \ldots, v_k$ could be all different. Then a test case is a combination of k test settings, one for each test factor (a k-tuple). The number of different test cases possible is the product of all k numbers $v_1, v_2, \ldots, v_k$ of test settings. For example, suppose out of nine test factors, if 3 have three test settings each, 4 have four test settings each, and 2 have five test settings each then the number of possible test cases is $3^3 4^4 5^2$ = 1,72,800. The exponential expression $3^3 4^4 5^2$ represents the combinatorial test structure and its expanded form 1,72,800 is the number of possible test cases. In many practical applications the number of possible test cases is too large to test them all. In combinatorial testing, combinatorial mathematics and computational methods are used to determine a small set (called test suite) of test cases which covers all test settings of each factor and all t-way combinations (t-tuples) of test settings for some $t \geqslant 2$. The value of t (called strength of the test suite) is chosen with the objective that the test suite will exercise the combinations corresponding to the faults for which the SUT could fail. Methods for specification of test factors, test settings, and the strength t are largely application domain specific and a subject of continuing research [4]. In this paper we address the problem of constructing (generating) the test suite after the test factors and the test settings have been specified.

Orthogonal arrays (OAs) are tabular arrangements of symbols which satisfy certain combinatorial properties. In the1960s and1970s Japan and starting in the 1980s in the US and Europe, Genichi Taguchi promulgated the use of OAs (of strength two) as templates for Design of Experiments (DoEs) [6–8]. In the late 1980s and early 1990s, inspired in part by Taguchi, some software engineers started investigating the use of OAs for pairwise (2-way) combinatorial testing of software and hardware–software systems. In Section 2, we review the use of OAs for DoEs and software testing. Soon the limitations of OAs to generate test suites for software testing became apparent. Covering arrays (CAs) are generalizations of OAs with slightly relaxed combinatorial properties. Covering arrays were found to be better suited than OAs for generating test suites for software testing. In Section 3 we discuss the use of CAs for pairwise (2-way) testing of software. Subsequent investigations of the reports of actual software failures showed that pairwise (2-way) testing is useful but it may not always be sufficient. Also, test factors and test settings are subject to various types of constraints imposed by the semantics of the SUT and the runtime environment. In Section 4, we discuss t-way combinatorial testing (CT) for $t \geqslant 2$ with support of constraints. Combinatorial testing for $t \geqslant 2$ is now practical because efficient tools for generating test suites for t-way testing with support of constraints have become available. A brief summary appears in Section 5.

2. Use of orthogonal arrays for design of experiments and software testing

The concept of Orthogonal arrays (OAs) was formally defined by Rao [9]. OAs are generalization of Latin squares [10]. The matrix shown in Table 1 is an orthogonal array (OA) referred to as OA(8, $2^4 \times 4^1$, 2). The first parameter (which is 8) indicates the number of rows and the second parameter (which is $2^4 \times 4^1$) indicates that there are five columns of which four have 2 distinct elements each, denoted here by {0,1}, and one column has 4 distinct elements, denoted here by {0,1,2,3}. The third parameter (which is 2) indicates that this OA has strength 2, which means that every set of two columns contains all possible pairs of elements exactly the same number of times. Thus every pair of the first four columns contains the four possible pairs of elements {00, 01, 10, 11} exactly twice and every pair of columns involving the fifth column contains the eight possible pairs of elements {00, 01, 02, 03, 10, 11, 12, 13} exactly once. In an OA of strength 3, every set of three columns contains all possible triplets of elements exactly the same number of times.

A fixed-value orthogonal array denoted by OA(N, v^k, t) is an $N \times k$ matrix of elements from a set of v symbols $\{0, 1, \ldots, (v-1)\}$ such that every set of t-columns contains each possible t-tuple of elements the same number of times. The positive integer t is the strength of the orthogonal array. In the context of an OA, elements such as $0, 1, 2, \ldots, (v-1)$ used in Table 1 are symbols rather than numbers. The combinatorial property is not affected by the symbols that are used for the elements. Every set of three columns of a fixed value orthogonal array of strength 2 represents a Latin square (one column representing the rows, one column representing the columns and the third column representing the symbols). A mixed-value orthogonal array is an extension of fixed-value OA where $k = k_1 + k_2 + \ldots + k_n$; k_1 columns have v_1 distinct elements, k_2 columns have v_2 distinct elements, etc., and k_n columns have v_n distinct elements, where $v_1, v_2, \ldots, v_k$ are different. Mathematics of OAs and extensive references can be found in [11]. Neil Sloane maintains an electronic library of known OAs [12].

The term design of experiments (DoEs) refers to a methodology for conducting controlled experiments in which a system is exercised (worked in action) in a purposeful (designed) manner for chosen test settings of various input variables (called factors). The corresponding values of one or more output variables (called responses) are measured

Table 1
Orthogonal array OA (8, $2^4 \times 4^1$, 2).

	1	2	3	4	5
1	0	0	0	0	0
2	1	1	1	1	0
3	0	0	1	1	1
4	1	1	0	0	1
5	0	1	0	1	2
6	1	0	1	0	2
7	0	1	1	0	3
8	1	0	0	1	3