Orthogonal Array application for optimal combination of software defect detection techniques choices

LJUBOMIR LAZIC[a], NIKOS MASTORAKIS[b]

[a]Technical Faculty, University of Novi Pazar
Vuka Karadžića bb, 36300 Novi Pazar, SERBIA
llazic@np.ac.yu http://www.np.ac.yu

[b]Military Institutions of University Education, Hellenic Naval Academy
Terma Hatzikyriakou, 18539, Piraeu, Greece
mastor@ieee.org

Abstract: - In this paper, we consider a problem that arises in black box testing: generating small test suites (i.e., sets of test cases) where the combinations that have to be covered are specified by input-output parameter relationships of a software system. That is, we only consider combinations of input parameters that affect an output parameter, and we do not assume that the input parameters have the same number of values. To solve this problem, we propose interaction testing, particularly an *Orthogonal Array* Testing Strategy (OATS) as a systematic, statistical way of testing pair-wise interactions. In software testing process (STP), it provides a natural mechanism for testing systems to be deployed on a variety of hardware and software configurations. The combinatorial approach to software testing uses models to generate a minimal number of test inputs so that selected combinations of input values are covered. The most common coverage criteria are two-way or pairwise coverage of value combinations, though for higher confidence three-way or higher coverage may be required. This paper presents some examples of software-system test requirements and corresponding models for applying the combinatorial approach to those test requirements. The method bridges contributions from mathematics, design of experiments, software test, and algorithms for application to usability testing. Also, this study presents a brief overview of the response surface methods (RSM) for computer experiments available in the literature. The Bayesian approach and *orthogonal arrays* constructed for computer experiments (OACE) were briefly discussed. An example, of a novel OACE application, to STP optimization study was also given. In this case study, an *orthogonal array* for computer experiments was utilized to build a second order response surface model. Gradient-based optimization algorithms could not be utilized in this case study since the design variables were discrete valued. Using OACE novel approach, optimum combination of software defect detection techniques choices for every software development phase that maximize all over Defect Detection Effectiveness of STP were determined.

Key-Words: - Software testing, Opzimization, Design of Experiments, Orthogonal array

1 Introduction

Many IT organizations struggle with how to determine the proper balance of testing in light of business demands and budgetary limitations. Testing has historically been difficult to optimize because of a number of factors. IT has always faced the constraints of balancing time, quality and cost. Combine this with the reality that it is often difficult to measure exactly how effective testing is for a given application, and you can easily see that many formal testing processes are far from optimal. Yet many IT organizations are not sure how to make them better.

So what exactly is optimized testing, and how does it differ from the quality control and quality assurance practices that many IT organizations currently have in place? Simply put, optimized testing is a practical approach to improving application quality that balances quality, cost and schedules to prioritize testing and optimize limited resources. The efficiency of this practical testing approach ensures that the finite testing resources at IT management's disposal are used to their most productive levels while eliminating harmful defects and errors during the testing process. Optimized testing practices, methods and tools now exist that allow the QA team to better align testing activities with business requirements by prioritizing testing activities based on two key factors: importance to the business and risk to the

business [25,26]. Business users, development, QA and other key constituents can now collaborate to prioritize requirements and identify and prioritize project risks. And, when required, they can collaborate to resolve problems during the test execution. Once priorities have been established, QA can confidently select the optimum set of tests to thoroughly test the high-priority areas of an application and adequately test all other areas [25,26]. This paper presents a novel OACE approach for software testing process (STP) optimization study finding optimum combination of software defect detection techniques (DDT) choices for every software development phase that maximize all over Defect Removal Effectiveness (DRE) of STP. The optimum combination of software defect detection techniques choices were determined applying *orthogonal arrays* constructed for post mortem designed experiment with collected defect data of a real project [27]. By integrating this unique approach to requirements management with test management and automation tools, tests and test suites can be automatically created and executed as in our Integrated and Optimized Software Testing Process (IOSTP) [25,27].

Software testing remains an important topic in software engineering. Testing efficiency and effectiveness, two goals which are not always in alignment, are both significantly improved. A recent report generated for the National Institute of Standards and Technology (NIST) found that software defects cost the U.S. economy 59.9 billion dollars annually [28]. While current technologies cannot hope to remove all errors from software, the report goes on to estimate that 22 billion dollars could be saved through earlier and more effective defect detection.

Black-box testing is a type of software testing that ensures a program meets its specification from a behavioral or functional perspective. The number of possible black-box test cases for any non-trivial software application is extremely large. The challenge in testing is to reduce the number of test cases to a subset that can be executed with available resources and can also exercise the software adequately so that majority of software defects are exposed. One popular method for performing black-box testing is the use of combinatorial covering designs [3-11] based on techniques developed in designing experiments. These designs correspond to test suites (i.e., a set of test cases) that cover, or execute, combinations of input parameters in a systematic and effective way, and are most applicable in testing data-driven systems where the manipulation of data inputs and the relationship between input parameters is the focus of testing. As pointed out by Dunietz, et al. [10], a technical challenge that remains in applying this promising technique in software testing is the construction of covering designs. There are two issues that need to be considered: the first and perhaps more important one is the size of the covering design since it dictates the number of test cases, and consequently, the amount of resources needed to test a software system; the second is the time and space requirements of the construction itself.

A great deal of research work has been devoted to generating small test suites [2, 3, 7, 12-16, 25]. Most researchers focus on uniform coverage of input parameters with uniform ranges; i.e., they consider test suites that cover all t -wise combinations of the input parameters for some integer t and the input parameters are assumed to have the same number of values. While such test suites apply to a large number of situations, in practice not all t -wise combinations of input parameters have equal priority in testing [3, 25], nor are all parameter domains of the same size. Testers often prioritize combinations of input parameters that influence a system's output parameters over those that do not [3, 15, 20]. Determining which set of input parameters influence a system's output parameters can be accomplished using existing analyses [3, 25] or can be discovered in the process of determining the expected result of a test case.

Covering all pairs of tested factor levels has been extensively studied. Mandl described using orthogonal arrays in testing of a compiler [16]. Tatsumi, in his paper on Test Case Design Support System used in Fujitsu Ltd [22], talks about two standards for creating test arrays: (1) with all combinations covered exactly the same number of times (orthogonal arrays) and (2) with all combinations covered at least once. When making that crucial distinction, he references an earlier paper by Shimokawa and Satoh [19]. Over the years, pairwise testing was shown to be an efficient and effective strategy of choosing tests [4-6, 10, 13, 25]. However, as shown by Smith et al. [20] and later by Bach and Shroeder [3] pairwise, like any technique, needs to be used appropriately and with caution. Since the probl em of finding a minimal array covering all pairwise combinations of a given set of test factors is NP-complete [14], understandably a considerable amount of research has gone into efficient creation of such arrays. Several strategies were proposed in an attempt to minimize number of tests produced [11]. Authors of these combinatorial test case generation strategies often describe additional considerations that must be taken into account before their solutions become practical. In many cases, they propose methods of handling these in context of their generation strategies. Tatsumi [22] mentions constraints as a way of specifying unwanted

combinations (or more generally, dependencies among test factors). Sherwood [18] explores adapting conventional t-wise strategy to invalid testing and the problem of preventing input masking. Cohen et al. [6] describe seeds which allow specifying combinations that need to appear in the output and covering combinations with mixed-strength arrays as a way of putting more emphasis on interactions of certain test factors. Kuhn et al measured the number of defects identified at different strengths of interaction coverage in software [13] and also in systems with embedded software [23]. Several other examples include application of experimental designs (DOE) to computer benchmarking, Object Oriented Testing, network testing, and compiler testing [24]. Indeed several of these mentioned studies have had an impact and tools for automatic construction of test suites have recently appeared. For instance, the Automated Efficient Test Generator tool (AETG) has been developed by Telecordia [6]; NASA has funded the development of the Test Case Generator tool (TCG) [23]; IBM has funded the Combinatorial Test Services tool (CTS) [1]; and TestCover.com has also introduced a web-based tool.

In this study response surface methods for computer experiments are investigated and some of the approaches available in the literature are discussed. The focus is on response surface model building using *orthogonal arrays* designed for computer experiments (OACE). Different Defect Detection Strategy and Techniques options, together with critical STP variables performance characteristics (e.g. DRE, cost, duration), are studied to optimize design, development, test and evaluation (DDT&E) cost using orthogonal arrays for computer experiments [25-27].

This paper is organized as follows. Section 2 presents the Best practices for optimized testing. Section 3 explain The Orthogonal Array Testing Strategy. A novel Orthogonal Arrays application as Design of Experiments Optimization Strategy are presented in Section 4. Finally, the paper is concluded in Section 5.

2 The optimized testing approach

2.1 Best practices for optimized testing

Adopting an optimized testing approach may sound overwhelming. However, the truth is that IT organizations can adopt optimized testing practices incrementally, implementing certain aspects tactically to achieve strategic advantage. This section offers some suggested ways to adopt an optimized testing approach. Optimized testing best practices include adoption of the application life cycle, requirements management, risk-based testing and automation [26]. This section highlights some suggested steps for incorporating these areas of optimized testing into your existing testing environment.

2.2 Adoption of an application quality life cycle

The cost of fixing defects increases exponentially as a defect moves through development into production. By adopting an application quality life cycle, quality is built in to the application from the earliest phases of its life cycle, rather than attempting to test it in when it's too late. This requires discipline in defining and managing requirements, implementing automated and repeatable best practices and access to the right information to make confident decisions. This best practice allows you to fix defects earlier, when there is more time to sufficiently address the problems and it is far less expensive. It lays the groundwork for continuous process improvement and higher-quality applications [25,28].

2.2.1 Testing Process Activities Flow

Once it was clear that Testing was much more than "Debugging" or "Problem fixing", it was apparent that testing was more than just a phase near the end of the development cycle. Testing has a life cycle of its own and there is useful and constructive testing to be done throughout the entire life cycle of development. This means that testing process begins with the requirements phase and from there parallels the entire development process. In other words, for each phase of the development process there is an important testing activity. This necessitates the need to migrate from an immature, ad hoc way of working to having a full-fledged Testing Process. The following is the life cycle for the complete Test Development and Execution Process scheme.

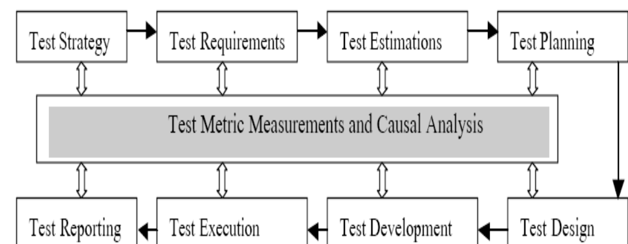

Fig. 1 Test Development and Execution Process scheme.

The specification defines the program correct behavior. The incorrect behavior is a software failure. It can be improper output, abnormal termination, and unmet time or space constraints. Failures are mostly caused by faults, which are missing or incorrect code. Error is a human action that produces a failure. An

abend is an abort termination of a program (like "blue screen of death" by Microsoft Windows). An omission is a required capability, which is not present in an implementation. Surprise is code that does not support a required capability. It can be surprising at code reuse. Bug is an error or a fault. The scope of STP is the collections of artifacts under test (AUT). Testing activities can be categorized by the scope of AUT that belongs to corresponding STP or SDL phase. Test artifact under test can be the software requirement (SRUT), High level design (HLDUT), Low Level Design (LLDUT), code being tested is called implementation under test (CUT), integration test (IUT) system under test (SUT), or in object-oriented environment class under test (CLUT), object under test (OUT), method under test (MUT). A test case defines the input sequence of data, the environment and state of AUT, and the expected result. Expected result is what AUT should generate, actual result is what was generated by run. An oracle produces expected results. An oracle can be an automated tool or human resource. A test is called to be passing if expected results and actual results are equal, otherwise it is called to be no pass or fail.

Test cases can be designed for positive testing or negative testing. Positive testing checks that the software does what it should. Negative testing checks that the software does not do what it should not do.

A test suit is a collection of test cases related to each other. Test run is the execution of a test suit. Test driver is a tool (can be a unit or utility program) that applies test cases to AUT. A stub is a partial, temporary implementation of a component. The following figure shows the systems engineering view of testing. Test strategy identifies the levels of testing, the methods, test detection techniques (DDT) and tools to be used. Test strategy defines the algorithm to create test cases.

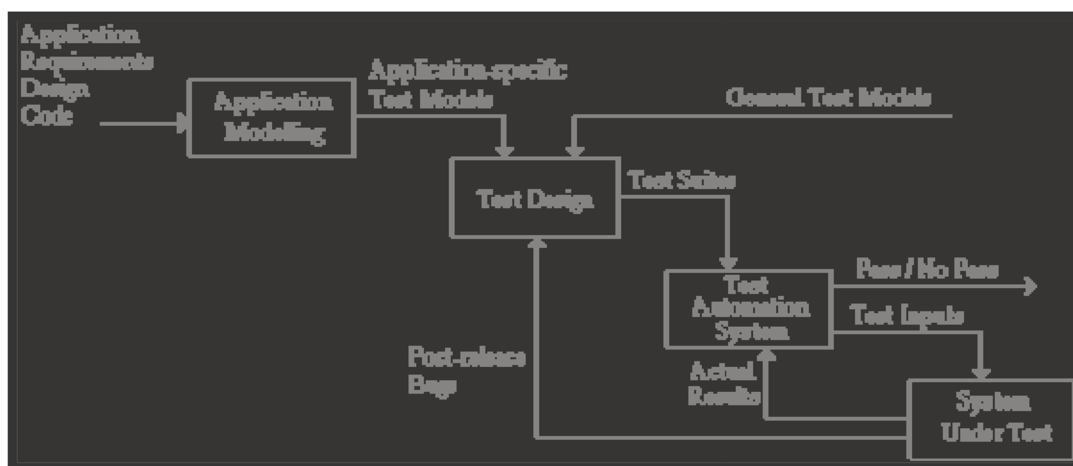

Fig. 2 The systems engineering view of testing

Test design produces test cases using a test strategy. Test effectiveness of DDT is the ability of the test strategy to find the bugs. Test efficiency is the cost of finding bugs. Strategies for the test design can be functional (Black-box), structural (White-box), hybrid (Gray-box) and fault-based. Functional testing is based on the specification of software, without knowing something about program code. It uses the specified or expected behavior. It is also called specification based, behavioral, and responsibility-based or black-box testing.

Structural testing relies on the structure of the source code to develop test cases. It uses the actual implementation to create test suits. It is also called implementation based, white box or clear box testing. Hybrid testing is the blend of functional and structural testing. It is also called gray-box testing.

Fault-based testing introduces faults into code (mutation) to see if these faults are revealed by a test suite. Regression testing is retesting the software with the same test cases. After a bug is fixed, the product should be tested at least with the bug revealer test case.

Coverage is the percentage of elements required by a test strategy that have been exercised by a test suite. There are many coverage models. Statement coverage is the percentage of source code statements executed at least once by a test suite. Clearly, statement coverage can be used only by structural or hybrid testing. Testing should make effort to reach 100% code coverage. This can avoid the user to run untested code. All these definitions raise a lot of questions and problems, and all of them cannot be dealt in this article (see references in [25]), although the most important ones can be found below. The testing strategy defines how test design should produce the test cases, but nothing it can tell us about how much testing is enough, and how effective the testing was. Test case effectiveness depends on numerous factors, and can be evaluated after the end of testing, which is normally too late. To avoid these problems, testers should perform in-process

evaluation of proposed and planned STP, according to established performance metrics and quality criteria [25,29] as we described below.

The testers should verify test cases at the end of test design, check the conformance of test cases to meet the requirements. It should also check the specification coverage. Validation is after test execution. Knowing the result, an effectiveness rate should count, and if it is under the threshold, the test suite should be analyzed, and the test process should be corrected.

A simple metric for effectiveness, which is only test suite dependent [29]. It is the ratio of bugs found by test cases (N_{tc}) to the total number of bugs (N_{tot}) reported during the test cycle (by test cases or by side effect):

$$TCE = 100 * N_{tc} / N_{tot} \ [\%] \quad (1)$$

This metric can evaluate effectiveness after a test cycle, which provides in-process feedback about the actual test suite effectiveness. To this metric a threshold value should create. This value is suggested to be about 75%, although it depends on the application.

When the *TCE* value is above the threshold, the test case can be said effective according to very useful model for dealing with defects as depicted on Fig. 3. If it is below, testers should correct the test plan, focusing on side effect bugs.

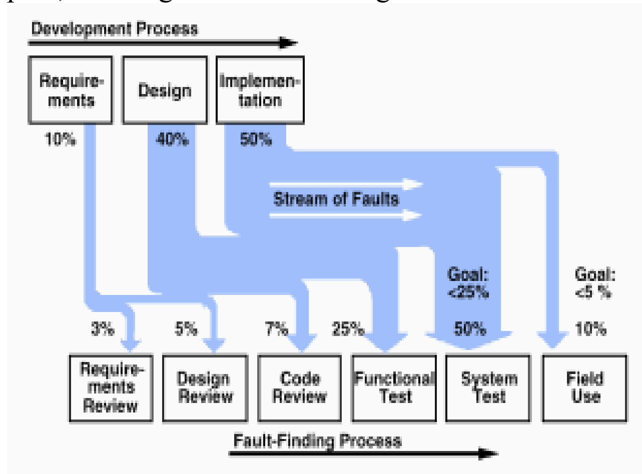

Fig. 3 Fault Injection Model Traditional

It basically says that given a software project – you have defects being "injected" into it (from a variety of sources) and defects being removed from it (by a variety of means). This high-level model is good to use to guide our thinking and reasoning about defects and defect processes. So, based on this model, the goal in software development, for delivering the fewest defects, is to: minimize the number of defects that go in maximize the number of defects that are removed.

2.2.2 Defect Removal Efficiency

A key metric for measuring and benchmarking the IOSTP [25] by measuring the percentage of possible defects removed from the product at any point in time. Both a project and process metric – can measure effectiveness of quality activities or the quality of a all over project by:

$$DRE = E/(E+D) \quad (2)$$

Where *E* is the number of errors found before delivery to the end user, and *D* is the number of errors found after delivery. The goal is to have *DRE close to 100%.* The same approach is applied to every test phase denoted wit *i* :

$$DRE_i = E_i / (E_i + E_{i+1}) \quad (3)$$

Where E_i is the number of errors found in a software engineering activity *i*, and E_{i+1} is the number of errors that were traceable to errors that were not discovered in software engineering activity *i*. The goal is to have this DRE_i approach to 100% as well i.e., errors are filtered out before they reach the next activity. Projects that use the same team and the same development processes can reasonably expect that the *DRE* from one project to the next are similar.

For example, if on the previous project, you removed 80% of the possible requirements defects using inspections, then you can expect to remove ~80% on the next project. Or if you know that your historical data shows that you typically remove 90% before shipment, and for this project, you've used the same process, met the same kind of release criteria, and have found 400 defects so far, then there probably are ~50 defects that you will find after you release. How to combine DDT to achieve high DRE, let say >85%, as a threshold for STP required effectiveness, is explained in section 4. which describe optimum combination of software defect detection techniques choices determination applying *orthogonal arrays* constructed for post mortem designed experiment with collected defect data of a real project [27].

2.3 Risk-based testing

An important area to focus on when optimizing your testing is identifying all of the business and technical requirements that exist for an application, and then prioritizing them based on the impact of failure on the business. QA teams should ensure they have access to the application's business and technical requirements in order to create effective test requirements. Involving business managers, test managers and QA architects will help achieve the balance of testing that is optimal [25-28]. The advantage of automated risk-based testing is that it adds a level of objectivity not available with traditional testing, where individual testers were left to determine what should be tested and when.

Thoroughly understanding and correctly prioritizing testing requirements can have the greatest impact on successful delivery of a high-quality application. By implementing an optimized testing solution, IT can ensure quality activities accurately reflect business priorities, and can make certain they are testing the right areas of an application within the constraints of the schedule. Using an optimized testing solution, the risks are calculated automatically and time estimates are rolled up per requirements balancing quality, schedule and cost through risk-based practices. This allows testers to apply a time factor to existing risk factors, which enables users to quickly select the highest-priority test cases and understand how long it will take to test them [26].

2.4 Software Testing Optimization Model and IT benefits

With optimized testing, IT organizations are able to balance the quality of their applications with existing testing schedules and the costs associated with different testing scenarios. Optimized testing provides a sound and proven approach that allows IT to align testing activities with business value. The practices, processes and tools that encompass optimized testing offer many benefits. The increasing cost and complexity of software development is leading software organizations in the industry to search for new ways through process methodology and tools for improving the quality of the software they develop and deliver.

2.4.1 Manage “what-if” scenarios -A Software Testing Optimization Model

Such scenarios are invaluable for determining where testing resources should be spent at the beginning of software development project. With an optimized testing solution, you can create what-if scenarios to help users understand the impact of changing risks, cycle attributes and requirements as priorities change. This insight proves invaluable when a testing organization is trying to determine the best way to balance quality with cost and schedule. By understanding the impact of different factors on testing, IT managers can identify the right balance.

We applied the End-to-End (E2E) Test strategy in our Integrated and Optimized Software Testing framework (IOSTP) [25-27]. End-to-End Architecture Testing is essentially a "gray box" approach to testing - a combination of the strengths of white box and black box testing. In determining the best source of data to support analyses, IOSTP with embedded RBOSTP considers credibility and cost of each test scenario i.e. concept. Resources for simulations and software test events are weighed against desired confidence levels and the limitations of both the resources and the analysis methods. The program manager works with the test engineers to use IOSTP with embedded RBOSTP [26] to develop a comprehensive evaluation strategy that uses data from the most cost-effective sources; this may be a combination of archived, simulation, and software test event data, each one contributing to addressing the issues for which it is best suited.

The central elements of IOSTP with embedded RBOSTP are: the acquisition of information that is credible; avoiding duplication throughout the life cycle; and the reuse of data, tools, and information. The system/software under test is described by objectives, parameters i.e. factors (business requirements - BR are indexed by j) in requirement specification matrix, where the major capabilities of subsystems being tested are documented and represent an independent i.e. input variable to optimization model. Information is sought under a number of test conditions or scenarios. Information may be gathered through feasible series of experiments (E): software test method, field test, through simulation, or through a combination, which represent test scenario indexed by i i.e. sequence of test events. Objectives or parameters may vary in importance α_j or severity of defect impacts. Each M&S or test option may have k models/tests called modes, at different level of credibility or probability to detect failure β_{ijk} and provide a different level of computed test event information benefit B_{ijkl} of experimental option for cell (i,j), mode k, and indexed option l for each feasible experiment depending on the nature of the method and structure of the test. Test event benefit B_{ijkl} of feasible experiment can be simple ROI or design parameter solution or both etc. The cost C_{ijkl}, of each experimental option corresponding to (i,j,k,l) combination must be estimated through standard cost analysis techniques and models. For every feasible experiment option, tester should estimate time duration T_{jikl} of experiment preparation end execution. The testers of each event, through historical experience and statistical calculations define the E_{ijkl}'s (binary variable 0 or 1) that identify options. The following objective function is structured to maximize benefits and investment in the most important test parameters and in the most credible options. The model maintains a budget, schedule and meets certain selection requirements and restrictions to provide feasible answers through maximization of benefit index - $B_{enefit}I_{ndex}$:

$$B_{enefit}I_{ndex} = \max_{i,j,k,l} \sum_j \sum_i \sum_k \sum_l \alpha_j \beta_{ijk} B_{ijkl} E_{ijkl} \quad (4)$$

Subject to:

$$\sum_j \sum_i \sum_k \sum_l C_{ijkl} E_{ijkl} \le BUDGET$$ (Budget constraint);

$\sum_{j}\sum_{i}\sum_{k}\sum_{l} T_{ijkl}E_{ijkl} \leq TIMESCHEDULE$ (Time-schedule constraint)

$\sum_{l} E_{ijkl} \leq 1$ for all *i,j,k* (at most one option selected per cell *i, j, k* mode)

$\sum_{k}\sum_{l} E_{ijkl} \geq 1$ for all *i,j* (at least one experiment option per cell *i, j*)

3 Orthogonal Array Testing Strategy (OATS) and Techniques

The Orthogonal Array Testing Strategy (OATS) is a systematic, statistical way of testing pair-wise interactions. It provides representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique particularly useful for integration testing of software components (especially in OO systems where multiple subclasses can be substituted as the server for a client). It is also quite useful for testing combinations of configurable options (such as a web page that lets the user choose the font style, background color, and page layout). Dr. Genichi Taguchi was one of the first proponents of orthogonal arrays in test design. His techniques, known as Taguchi Methods, have been a mainstay in experimental design in manufacturing fields for decades.

Orthogonal arrays are two dimensional arrays of numbers which possess the interesting quality that by choosing any two columns in the array you receive an even distribution of all the pair-wise combinations of values in the array. The method of orthogonal arrays is an experimental design construction technique from the literature of statistics. In turn, construction of such arrays depends on the theory of combinatorics. An orthogonal array is a balanced two-way classification scheme used to construct balanced experiments when it is not practical to test all possible combinations. The size and shape of the array depend on the number of parameters and values in the experiment. Orthogonal arrays are related to *combinatorial designs*. An orthogonal array is a balanced two-way classification scheme used to construct balanced experiments when it is not practical to test all possible combinations. The size and shape of the array depend on the number of parameters and values in the experiment.

Definition 1: Orthogonal array $O(\rho, k, n, d)$
An orthogonal array is denoted by $O(\rho, k, n, d)$, where:

- ρ is the number of rows in the array. The k-tuple forming each row represents a single test configuration, and thus ρ represents the number of test configurations.
- k is the number of columns, representing the number of parameters.
- The entries in the array are the values $0, \ldots, n-1$, where $n = f(n_0, \ldots, n_{k-1})$.

Typically, this means that each parameter would have (up to) n values.

- d is the *strength* of the array (see below).

An orthogonal array has *strength* d if in any $\rho \times d$ sub-matrix (that is, select any d columns), each of the n^*d possible d-tuples (rows) appears the same number of times (>0). In other words, all d-interaction elements occur the same number of times.

Here is some terminology for working with orthogonal arrays followed by an example array in Fig. 4 [24,25]:

- Runs - ρ: the number of rows in the array. This directly translates to the number of test cases that will be generated by the OATS technique.
- Factors - k: the number of columns in an array. This directly translates to the maximum number of variables that can be handled by this array.
- Levels - n: the maximum number of values that can be taken on by any single factor. An orthogonal array will contain values from 0 to Levels-1.
- Strength - d: the number of columns it takes to see each of the Levels^{Stength} possibilities equally often.
- Orthogonal arrays are most often named following the pattern $L_{Runs}(Levels^{Factors})$.

	Factors			
Runs	0	0	0	0
	0	1	1	2
	0	2	2	1
	1	0	1	1
	1	1	2	0
	1	2	0	2
	2	0	2	2
	2	1	0	1
	2	2	1	0

Fig. 4 An $L_9(3_4)$ orthogonal array with 9 runs, 4 factors, 3 levels, and strength of 2.

3.1 Facts and Industry experiences

Being intelligent about which test cases you choose can make all the difference between (a) endlessly executing tests that just aren't likely to find bugs and don't increase your confidence in the system and (b) executing a concise, well-defined set of tests that are likely to uncover most (not all) of

the bugs and that give you a great deal more comfort in the quality of your software.

Some advantage of the Orthogonal Array Testing Strategy (OATS) is outlined below:

• Pairwise testing protects against pairwise bugs while dramatically reducing the number of tests to perform which is especially cool because pairwise bugs represent the majority of combinatoric bugs and such bugs are a lot more likely to happen than the ones that only happen with more variables.

• Plus, the availability of tools means you no longer need to create these tests by hand.

• Pairwise testing might find some pairwise bugs while dramatically reducing the number of tests to perform, compared to testing all combinations, but not necessarily compared to testing just the combinations that matter, which is especially cool because pairwise bugs represent the majority of combinatoric bugs, or might not, depending on the actual dependencies among the variables in the product, and such bugs are more likely to happen than ones that only happen with more variables, or less likely to happen, because user inputs are not randomly distributed.

• Plus, the availability of tools means you no longer need to create these tests by hand, except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results.

What do all pairs do?

• Input is a set of equivalence classes for each variable.

• Sometimes the equivalence classes are those used in Boundary Value testing (min, min+, nominal, max-, max)

• Output is a set of (**partial**) test cases that approximate an orthogonal array, plus some pairing information.

All pairs assumptions are:

• Variables have clear equivalence classes.

• Variables are independent.

• Failures are the result of the interaction of a pair of variable values.

OATS provides a means to select a test set that:

• Guarantees testing the pair-wise combinations of all the selected variables.

• Creates an efficient and concise test set with many fewer test cases than testing all combinations of all variables.

• Creates a test set that has an even distribution of all pair-wise combinations.

• Exercises some of the complex combinations of all the variables.

• Is simpler to generate and less error prone than test sets created by hand.

As an example of the benefit of using the OATS technique over a test set that exhaustively tests every combination of all variables, consider a system that has four options, each of which can have three values. The exhaustive test set would require 81 test cases (3 x 3 x 3 x 3 or the Cartesian product of the options). The test set created by OATS (using the orthogonal array in Fig. 4) has only nine test cases, yet tests all of the pair-wise combinations. The OATS test set is only 11% as large at the exhaustive set and will uncover most of the interaction bugs. It covers 100% (9 of 9) of the pair-wise combinations, 33% (9 of 27) of the three-way combinations, and 11% (9 of 81) of the four-way combinations. The test set could easily be augmented if there were particularly suspicious three- and four-way combinations that should be tested. Interaction testing can offer significant savings. Indeed a system with 20 factors and 5 levels each would require 5^{20} = 95 367 431 640 625 i.e. almost 10^{14} exhaustive test configurations. Pair-wise interaction testing for 5^{20} can be achieved in 45 tests.

3.2 How to use this technique

The OATS technique is simple and straightforward. The steps are outlined below.

1. Decide how many independent variables will be tested for interaction. This will map to the Factors of the array.
2. Decide the maximum number of values that each independent variable will take on. This will map to the Levels of the array.
3. Find a suitable orthogonal array with the smallest number of Runs. A suitable array is one that has at least as many Factors as needed from Step 1 and has at least as many levels for each of those factors as decided in Step 2.
4. Map the Factors and values onto the array.
5. Choose values for any "left over" Levels.
6. Transcribe the Runs into test cases, adding any particularly suspicious combinations that aren't generated.

3.3 OART application areas in software testing

OART can be applied to Black-box testing strategy as outlined below:

• Unit testing: derive test cases for black-box testing of single components,

• System testing: derive test cases for black-box testing of entire systems,

• Known that significant number of failures is caused by parameter interactions that occur in typical, yet realistic situations,

• Assume that tests maximizing the interactions between parameters will find more faults,

• Test at least for all two-way interactions among all input parameter combinations because exhaustive testing (i.e. executing test cases for *all* possible input parameter combinations) cannot be afforded,
• Assume that the risk of an interaction failure among three or more input parameters is balanced against the ability to complete testing within a reasonable budget,
• Calculate the minimal set of test parameter combinations that test each pair-wise parameter combination.

OART can be applied to Configuration Testing as outlined below:
• Testing of complex systems with multiple configurations,
• Interoperability testing,
• Web testing,
• Known that faulty interaction between system components is a common source of system failures,
• Re-use existing suite of (system) test cases,
• Test at least for all two-way interactions among various system components because exhaustive testing (i.e. executing a suite of test cases for *all* possible configurations) cannot be afforded,
• Assume that the risk of an interaction failure among three or more components is balanced against the ability to complete testing within a reasonable budget,
• Calculate the minimal set of test configurations that test each pair-wise combination of components.
Which test input data shall be selected and what is benefit of OATS technique?

• Intelligent test case generation is vital to cut down costs and improve the quality of testing,
• Dramatically reduced overall number of test cases compared to exhaustive testing,
• Detects *all* faults due to a *single* parameter input domain,
• Detects *all* faults due to *interaction of two* parameter input domains,
• Detects *many* faults due to interaction of *multiple* parameter input domains.

Case studies [23,25,30] give evidence that the approach compared to conventional approaches is:
• more than twice as efficient (measured in terms of detected faults per testing effort) as traditional testing,
• about 20% more effective (measured in terms of detected faults per number of test cases) as traditional testing,
• Approach is applicable:
  - to generate detailed test case input data during unit testing
  - to generate high-level test cases during system testing
  - to generate test cases during configuration testing
  - seamlessly with conventional test methods.

For illustration let us analyze this example of Configuration Testing of communication system under test that has 4 components, each of which has 3 possible elements as depicted in Fig.5 .

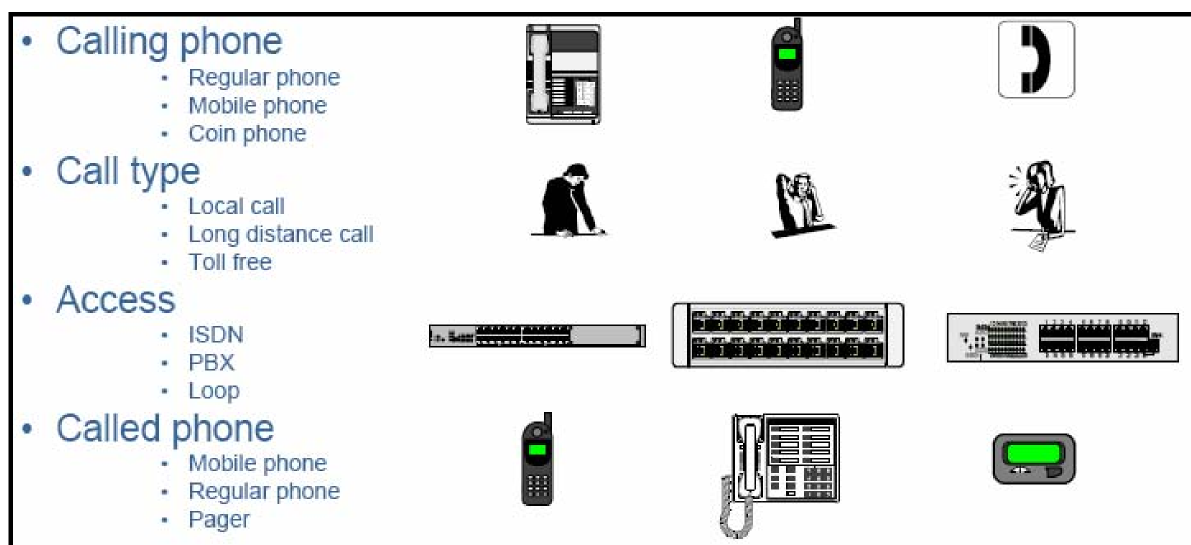

Fig. 5 Example of Configuration Testing of communication system

Those **4** components are: Calling phone (Regular phone, Mobile phone, Coin phone), Call type (Local call, Long distance call, Toll free), Access (ISDN, PBX, Loop), and Called phone (Mobile phone, Regular phone, Pager), each of which has 3 quoted possible elements.
Suite of system test cases exists and has to be executed with overall **81** = **3**x**3x3x3,** number of possible configurations. *But, with applying OART*

algorithm according to Fig. 4, requires only 9 test configurations instead of 81 that are already sufficient to cover all pair-wise component interactions.

4 A novel Orthogonal Arrays application as Optimization Strategy

Experimental optimization can be carried out in several ways. Most popular is the one-variable-at-a-time approach. This approach is however extremely inefficient in locating the true optimum when interaction effects are present. Multivariable design of experiments are since many years used to overcome the problems with interaction effects. There are two general groups of designs to choose from: Sequential or simultaneous experiment designs. The choice depends of the purpose of the study.

Are these both approaches, sequential and simultaneous design of experiments, competing alternatives or can they be joined into a comprehensive and effective optimization and model-building strategy? Our a novel Orthogonal Arrays application, may have come up with the answer. In this approach is first to optimize and then to study variable effects, significance, etc. (i.e. model-building).

In the past, optimization usually required answers to three ordered questions:

1. What variables (in our case, which Defect Detection Techniques – DDT) are the most significant?
2. In what way (in our case, find optimum DDT combination) do they affect the quality (in our case, effectiveness of software test activities with Defect Removal Efficiency – DRE) of the product or process?
3. What is the optimal combination of settings for these significant variables (in our case, DDT)?

4.1 DDT evalouation or Measurement Choice like customer satisfaction index

Before, Orthogonal Arrays application as Optimization Strategy – OAOS, we did DDT evaluation like customer satisfaction index to answer the 1. question above i.e. which Defect Detection Techniques – DDTs are the most significant. Every tester in test team assess 5 most frequently used DDT: DDT_1= Inspection – DBR, DDT_2= PBR, DDT_3= CEG+BOR+MI, DDT_4= M&S, DDT_5= Hybrid (Category Partition, Boundary value analysis,…., Path testing etc.), as briefly described in this section, according to nine Performance and Quality criteria given in Table 1. We have chosen a unified ordinal scale for the empirical Performance and Quality criterion from *1 (worst)* to *5 (best)* of the tester satisfaction level (TS level). This aspect is at most indeterminate in our approach.

Table 1. DDT Performance and Quality criteria for tester's assessment

Criterion C_k	Name	Description
1	**Overall**	Considering all aspects of the experience, how would you rate your overall level of satisfaction with DDT e.g. Combinatorial Test Services?
2	**Capability**	How satisfied are you that DDT e.g. Combinatorial Test Services has the functions and features to perform as expected?
3	**Usability**	How satisfied are you with the "ease of use" of DDT e.g. Combinatorial Test Services?
4	**Performance**	How satisfied are you with the response time or speed with which DDT e.g. Combinatorial Test Services executes its functions?
5	**Reliability**	How satisfied are you with the frequency, number, and seriousness of errors in DDT e.g. Combinatorial Test Services?
6	**Installation**	How satisfied were you with the ease of installation, initialization, and migration of DDT e.g. Combinatorial Test Services?
7	**Maintenance**	How satisfied are you with getting updates from alphaWorks, or patches from the development team for DDT e.g. Combinatorial Test Services?
8	**Information**	How satisfied are you with the accuracy, completeness, and time it takes to find information (online help, etc.) for DDT e.g. Combinatorial Test Services?
9	**Service**	How satisfied are you with the effectiveness of the DDT support e.g. alphaWorks discussion forum for Combinatorial Test Services, and responses to your e-mails?

The three Static Analysis test techniques are all based on Behavior Analysis which enables analysts to convert narrative text into a tabular representation that are DDT candidates: *Defect-based reading – DBR as* DDT_1, *perspective-based reading-PBR as* DDT_2 *,and Simulation and Model Testing of the Design prior to Implementation – M&S* as DDT_4. In its own right, Behavior Analysis helps to find errors

in requirements because it demands the analyst adopt a systematic approach to extracting functions, conditions and responses from the requirements text before synthesizing elementary test cases which can be validated individually. Even if the three test techniques are not used, Behavior Analysis is an effective method for finding errors in requirements and other software development documents (R&D D). The technique is easy to learn and implement, and can be applied to requirements documents of any size by selecting only critical sections for analysis.

Behavior Analysis is a pre-requisite for the three test methods; inspection, walkthrough and animation, and each test technique have its own area of applicability and benefits:

- test by inspection is useful where traditional inspections would be too difficult, time-consuming or expensive to implement in the user community
- test by scenario walkthrough is useful where the fit between proposed system and new or changed business process is a key consideration
- test by animation is most useful where requirements are not stable, where the system is aimed at individual users or where a more controlled Prototyping technique is required.

Errors in requirements present a most difficult challenge. Developers find it almost impossible to detect errors in requirements without the help of users. New methodologies involve users much more intimately, but these techniques are successful in specific situations where the risk of errors is low and overall project size is small. Traditional developments of larger systems still use a staged, rather than an iterative approach. Users are intimately involved only in the earliest stages and at the very end. The risk of requirements errors is extremely high in such projects, but in most cases requirements are not adequately tested. Many of these projects fail in the end, when the cost is highest, because they could not deliver the required business benefits. Testing of requirements and other software development documents (R&D D) is potentially the most valuable testing we can do, because errors in requirements are usually the most expensive to correct later, and present the biggest threat to the project's success. Behavior Analysis, testing by inspection, testing by walkthrough and testing by animation offer some hope that requirements can be `got right first time'.

A Cause-Effect Graphing – CEG+BOR+MI [15] is used as DDT_3. The Cause-Effect Graphing technique was invented by Bill Elmendorf of IBM in 1973. Instead of the test case designer trying to manually determine the right set of test cases, he/she models the problem using a cause-effect graph, and the software that supports the technique [15], calculates the right set of test cases to cover 100% of the functionality. The cause-effect graphing technique uses he same algorithms that are used in hardware logic circuit testing. Strict test case design in hardware ensures virtually defect free hardware. The starting point for the Cause-Effect Graph, applied to software testing, is the requirements document. The requirements describe what the system is intended to do. The requirements and other software development documents can describe real time systems, events, data driven systems, state transition diagrams, object oriented systems, graphical user interface standards, etc. A specification based testing strategy, called CEG-BOR [15], combines the use of cause-effect graphs (CEGs) as a mechanism for representing specifications and the use of the Boolean operator (BOR) strategy for generating tests for a Boolean expression. If all causes of a CEG are independent from each other, a test set for the CEG can be constructed such that all boolean operator faults in the CEG can be detected and the size of this test set grows linearly with the number of nodes in the CEG. Four case studies are conducted to provide empirical data on the performance of CEG-BOR [15]. Empirical results indicate that CEGs can be used to model a large class of software specifications and that CEG-BOR is very effective in detecting a broad spectrum of faults. Also, a BOR test set based on a CEG specification provides better coverage of the implementation code than test sets based on random testing, functional testing, and state-based testing. For a CEG that does not have mutually independent causes, the BOR strategy does not perform well. To remedy this problem, a new test generation strategy is presented, which combines the BOR strategy with the Meaningful Impact (MI) strategy, a recently developed test generation strategy for Boolean expressions. This new strategy, called BOR+MI [15], decomposes a Boolean expression into mutually independent components, applies the BOR or MI strategy to each component for test generation, and then applies the BOR strategy to combine the test sets for all components. The size and fault detection capability of a BOR+MI test set are very good. Both analytical and empirical results show that the BOR+MI strategy generates a smaller test set than the MI strategy and provides comparable fault detection ability as the MI strategy. This extension, called BRO+MI, detects incorrect relational operators in relational expressions and also accounts for user-defined or implicit restrictions on the causes of a CEG.

Traditional Test Case Design Techniques, including Equivalence Class Partitioning, Boundary Value Analysis and some White-box (statement, branch and path covering) are combined in Hybrid or Gray-box

DDT that we denoted as DDT_5 . These techniques rely on the test case designer to manually work out the proper combinations of test cases. Often, the test case designer does not use a formal test case design technique and relies on his/her "gut feel" to assess whether test coverage is sufficient. While these techniques do generate combinations of test cases, they often fall short on providing full functional coverage. Too often the normal flow or "go path" functionality has overlapping, redundant test cases, while exceptions and error conditions go untested. The Full-Lifecycle IOSTP methodology is a collection of testing techniques to verify and validate broad types of software products. The IOSTP uses a wide variety of techniques (described in [25]) that are available to deploy throughout all aspects of software development. The list of techniques is not meant to be complete – instead the goal is to make it explicit wide range of options available for STP optimization.

4.2 Application of the Borda optimal positional voting method to DDT ranking

The Borda method is used in the Risk Matrix software application [31] to rank risks from most-to-least critical on the basis of multiple evaluation criteria. We adapted this Borda method, on similar way, to rank all used Defect Detection Techniques (DDT) through software development life cycle from most-to-least performance and quality characteristics of DDT in revealing software faults (bugs, errors). This section describes in detail how the Borda method is applied, using the sample of DDT Assessment Entries Worksheet that every tester in test team provided as an illustration.

On the other hand, it is necessary to map the possible metrics values to the ordinal scale of the empirical criterion. We have chosen a unified ordinal scale for the empirical criterion from *1 (worst)* to *5 (best)* of the tester satisfaction level (TS level). This aspect is at most indeterminate in our approach. Hence, its tool support requires high flexibility for the adjustment or tuning of the measurement process of the customer satisfaction determination.

Borda proposed the following voting method in 1770: Given N - DDT candidates, if points of N -1, N - 2, . . . , and 0 are assigned to the first-ranked, second-ranked, . . . , and last-ranked candidate in each test team voter's preference order, then the winning candidate is the one with the greatest total number of points. Instead of voters, suppose that there are multiple criteria. If r_{ik} is the rank of alternative i - particular DDT, under criterion *k = 1 to 9 (DDT criteria from Table 1)*, the Borda count for alternative i is

$$b_i = \sum_{k=1}^{9} (N - r_{ik}) . \qquad (5)$$

The alternatives are then ordered according to these counts. The Borda method is an example of a *positional voting method*, which assigns P_j points to a voter's j^{th}-ranked candidate, $j = 1, \ldots, N$, and then determines the ranking of the candidates by evaluating the total number of points assigned to each of them. Voting theorists [31] have shown that the Borda method is the optimal positional voting method with respect to several standards, such as minimizing the number and kinds of voting paradoxes. In addition, if ties are not present in the criteria rankings, it is demonstrated that the Borda method is equivalent to determining the consensus rankings that minimize the sum of the squared deviations from the criteria rankings. The Borda method has been used to rank alternatives in a variety of applications, including a cost and operational effectiveness analysis (COEA) and an aircraft maintenance study. In the DDT ranking application, let N be the total number of DDTs, and the index i denote a particular DDT. Let the DDT criterion of **Overall** (from Table 1) assessment be denoted by k = 1, and the DDT criterion of **Capability** assessment be denoted by k = 2 etc. The rest of this section describes how the Borda voting method is implemented in our DDT rank assessment case.

4.2.1 Evaluate Rank of Each DDT with Respect to *Overall* criterion

Let J be the total number of possible **Overall** assessments. As discussed above, a DDT can be assessed by tester, as for the empirical criterion from *1 (worst)* to *5 (best)* of the tester satisfaction level (TS level), and so there are J = 5 possible assessments. Let Q_j be the j-th possible **Overall** assessment, which is assumed to be ordered in the following way: Q_j has a higher **Overall** point than Q_{j+1}. Thus, Q_1 = 1 (TS level), Q_2 =2 (TS level), etc. Let M_j be the number of DDTs having Q_j as the **Overall** rating. Table 2 gives the values of M_j that correspond to the sample given by testers.

Table 2 Values of Q_j, M_j, and T_j for sample given by testers

j	Qj	Mj	*Tj*
1	5 *(best)*	2	1.5
2	4	3	4
3	3	0	N/A
4	2	0	N/A
5	1 (*worst)*	0	N/A

Let T_j be the rank position for all DDTs that are given the j-th possible impact assessment.

How can we evaluate this rank position? The basic approach is to evaluate the rank of a tied alternative

as the average of the associated rankings. The following is a key result: if a is the first term in an arithmetic progression, t is the final term, and n is the number of terms, then $(n/2)(a + t)$ is the sum of the n terms. Because there are M_1 *DDTs* that are tied for positions 1 through M_1 , the sum of these rank positions is $(M_1 / 2)(1 + M_1)$. Thus, the average of this sum is $T_1 = (1/2)(1 + M_1)$. Similarly, there are M_2 DDTs that are tied for positions $M_1 + 1$ through $M_1 + M_2$, so that the average of this sum is $T_2 = (1/2)(2M_1 + 1 + M_2)$. More generally, if $M_j > 0$,
$T_j = 1/2(2C_j + 1 + M_j)$, where

$$C_j = \sum_{r=1}^{j-1} M_r \qquad (6)$$

for $j > 1$ and $C_1 = 0$. The values of T_j are given in Table 2 for the sample given by testers.
Let r_{i1} be the rank of the i- th DDT with respect to the impact assessment. If the i-th DDT has the j-th possible impact assessment, then set $r_{i1} = T_j$. The values of r_{i1} are given in Table 3 for the sample given by testers.

Table 3 Borda Points and Count for sample given by testers

DDT No.	C_1 Criterion	C_2 Criterion	r_{i1}	r_{i2}	Borda Count	Borda Rank
1	5	3	1.5	3.5	5	0
2	4	5	4	1	5	0
3	4	4	4	2	4	3
4	5	3	1.5	3.5	5	0
5	4	2	4	5	1	4

4.2.2 Evaluate Rank of Each DDT with Respect to *Capability* criterion

Let H be the total number of possible **Capability** assessments. As discussed above, there are five default **Capability** ranges and so $H = 5$. Let P_h be the highest **Capability** associated with the h-th possible assessment, and let these be ordered such that $P_h > P_{h+1}$. Let N_h be the number of DDTs that are assigned the h-th possible **Capability** assessment. Table 4 shows the values of P_h and N_h that are used for our numerical example, where the values of N_h are derived from sample given by testers.
Let S_h be the rank position for all DDTs that are given the h-th possible **Capability** assessment.
As before, if $N_h > 0$,

$S_h = 1/2(2B_h + 1 + N_h)$, where

$$B_h = \sum_{r=1}^{h-1} N_r \qquad (7)$$

for $h > 1$ and $B_1 = 0$. The values of S_h are given in Table 4 for the sample given by testers.

Table 4 Values of P_h, N_h, and S_h for Sample given by testers

h	P_h	N_h	S_h
1	5	1	1
2	4	1	2
3	3	2	3.5
4	2	1	5
5	1	0	N.A.

Let r_{I2} be the rank of the ith DDT respect to the **Capability** of occurrence. If the ith DDT has the h-th possible assessment, then set $r_{I2} = S_h$. The values of r_{I2} are given in Table 3 for the sample given by testers.

4.2.3 Determine Borda Ranking of Each DDT

Let N be the total number of DDTs, which satisfies

$$N = \sum_{h=1}^{H} N_h \qquad (8)$$

The Borda Count for DDT i is computed with formulae:

$$b_i = (N - r_{i1}) + (N - r_{i2}) \qquad (9)$$

The final step is to rank the DDTs with respect to their Borda Count. In particular, the DDT with the highest Borda Count is the best DDT according to testers Performance and Quality multi-criteria assessment, the DDT with the second highest count is the next DDT with highest score, and so forth. The Borda Rank for given DDT is the number of other DDTs that are better then that DDT. Table 3 provides both the Borda Count and Borda Rank for the sample given by testers. Defect Detection Techniques DDT_1, DDT_2, and DDT_4 are tied with the highest Borda Count, and so their Borda Rank is *0*. DDT_3 has a Borda Rank of *3,* because there are three other DDTs that are more critical. DDT_5 has a Borda Rank of *4,* because there are four other DDTs that are better then that DDT_5. The foregoing algorithm has been implemented as part of the software application. The same procedure is accomplished for the rest 7 criteria and Borda Rank for the sample given by testers wasn't changed i.e. were the same.
As a conclusion after Borda Ranking of DDT candidates, we did DDT evaluation like testers satisfaction index to answer the 1. question from optimization point of view i.e. which Defect Detection Techniques – DDTs are the most significant? According to testers assessment of 5 most frequently used DDT in IOSTP [25]: DDT_1= Inspection – DBR, DDT_2= PBR, DDT_3= CEG+BOR+MI, DDT_4= M&S, DDT_5= Hybrid

(Category Partition, Boundary value analysis, Path testing etc.) *three* of DDTs have the highest rank 0 i.e. $DDT_1=DDT_2=DDT_4=0$, then $DDT_3=$ CEG+BOR+MI is next ranked and the last was DDT_5. Because of that we will group those three DDT with highest rank 0, call them Static Test Techniques – TT1 and treat all three DDTs as *one factor* in optimization experiment applying Orthogonal Arrays as Optimization Strategy. Next high Borda ranked $DDT_4=$ CEG+BOR+MI we designate with TT2 and the last ranked DDT_5 as TT3.

In next section we provided the answers to 2. and 3. optimization question given above i.e. in what way DDTs combination do affect the quality (in our case, effectiveness of software test activities with Defect Removal Efficiency – DRE) of the software testing process and what is the optimal combination of DDTs for these significant variables (in our case - TT1,TT2 and TT3)?

4.3 Response Surface Model Building Using Orthogonal Arrays

Multidisciplinary design optimization (MDO) is an important step in the conceptual design and evaluation of STP effectiveness and efficiency since many factors has a significant impact on performance and software development lifecycle (SDL) cost. The objective in MDO is to search the software design and STP space efficiently to determine the values of design and process variables that optimize performance characteristics subject to system constraints.

An alternative is to utilize *response surface methodology* (RSM) to obtain mathematical models that approximate the functional relationships between performance characteristics and design/process variables. A common approach used in RSM is to utilize central composite designs (CCD) from the design of experiments literature to sample the SDL and STP space efficiently [25,27]. With this approach, design analyses (experiments) are performed at the statistically selected points specified by a CCD matrix. The resulting data is used to construct response surface approximation models using least squares regression analysis. These response surface equations are then used for MDO and for rapid sensitivity studies.

However, like most experimental designs, CCD is designed with the physical experiments in mind where the dominant issue is the variance of measurement of the response. In a physical experiment, there is usually some variability in the output response with the experiment repeated with the same inputs. In contrast, the output of computer (software testing) experiments is (in almost all cases) deterministic. Generally, there is no measurement error or no variability in analysis outputs. Therefore, experimental designs constructed to minimize variability of measurements may not be the best choice for computer experiments [17].

In this study response surface methods for computer experiments are investigated and some of the approaches available in the literature are discussed. The focus is on response surface model building using *orthogonal arrays* designed for computer experiments (OACE).

4.3.1 Response Surface Model Building Using Central Composite Designs

Response surface methods (RSM) can be utilized for MDO in cases where computerized design tool integration is difficult and design effort is costly. The first step in RSM, is to construct polynomial approximations to the functional relationships between design or process variables and performance characteristics (e.g. DDT, DRE) [25,27]. In the next step, these parametric models are used for MDO and to determine variable sensitivities. A quadratic approximation model in the form given below (10) is commonly used since it can account for individual parameter effects, second-order curvature or non-linearity (square terms), and for two-parameter interactions (cross terms).

$$\hat{y} = b_0 + \sum_{i=1}^{k} b_i X_i + \sum_{i\neq j,1}^{k} b_{ij} X_i X_j + \sum_{i=1}^{k} b_{ii} X_i^2$$

(10)

where : $\hat{y}$ - approximation of output variable i.e. trial response (the performance characteristic to be optimized), k- number of factors, b_0 ,b_{ij} – are estimated least squares regression coefficients, based on the design and analysis data obtained by sampling the design/process space (or by conducting experiments), $X_i = \frac{x_i - x_{i0}}{\Delta x}$ coded i^{th} factor values, x_i–real i^{th} factor values, x_{i0} -real factor value in "**NULL**" point (point in experimental center) and Δx- variation interval.

In some cases, however, RSM using CCD may not result in a good representation of the response surface as may be evidenced by poor predictions of the design analysis results. The reasons for this problem can be mainly due to;

1) The response surface is more complex than can be represented by a second order approximation model given by

equation (10),

2) There are other influential design/process variables and interactions other than those currently under study,
3) The sample design points (experiments) specified by a CCD may not be suitable in terms of selection of these specific points for experimentation with computerized design analysis tools.

The third problem is directly related to the choice of specific experimental design points. In order to address this problem and to improve response surface model building using computer experiments, a study was conducted.

4.3.2 Response Surface Model Building Methods for Computer Experiments

Bayesian approach to experimental design appears to be a growing area of research. However, the application of Bayesian experimantal design methods in real design analysis and optimization problems have been limited partly due to the lack of user friendly software [17]. Further development appears to be needed before they can be applied to practical design optimization problems.

The frequentist approach, surveyed by Owen [17] on the other hand, introduces randomness by taking function values that are partially determined by pseudorandom number generators. Then this randomness is propagated through to randomness in the estimate. Owen, lists a set of randomized orthogonal arrays for computer experiments. The Statlib computer programs (http://lib.stat.cmu.edu/designs/) to generate these orthogonal arrays are also listed.

The use of these orthogonal arrays in practice for response surface model building would be similar to utilizing central composite designs, with a potential of improving model accuracy for computer experiments. In the following section, an example application to an optimum DDTs combination selection and optimization study for an Integrated and Optimized Software Testing Process IOSTP [25] is presented.

4.3.3 Example Application: optimum DDTs combination selection and optimization study for an IOSTP

Traditionally, the objective in a MDO study has been to search the design space to determine the values of design variables (such as DDTs) that optimize a performance characteristic (such as DRE) subject to software testing process constraints. However, research shows that up to 82% of the life software testing cycle cost is committed during the early design phase [25,29]. Therefore, significant cost savings could be realized if designers and test managers were better able to evaluate their designs on a cost basis.

This study focuses on rapid multidisciplinary analysis and evaluation-on-a-DRE maximum-basis for DDT combination choices selection for each test phase activities i.e. P1- software requirement (SRUT), P2- High level design (HLDUT), P3- Low Level Design (LLDUT), P4- code under test (CUT), P5- integration test (IUT), P6- system under test (SUT) and finally P7- Acceptance test, recall section 2.2.1. Different Defect Detection Strategy and Techniques options, together with critical STP variables performance characteristics (e.g. DRE, cost, duration), are studied to optimize design, development, test and evaluation (DDT&E) cost using orthogonal arrays for computer experiments [25-27]. Calculus-based optimizers could not have been used in this case since material and technology options selection require the study of design variables that have discrete values. This study has the following steps:

1. Identify the design variables to be studied and alternative levels

In this study, design of maximum DRE percentage of STP optimization problem solving with best DDT choice combination in each phase P1 to P7 as controlled variables values is determined by designed experiment plan using *orthogonal arrays* designed for this computer experiment (OACE). To simplify the analysis such as decreasing factor's values (only three DDT number) applying Borda Ranking of DDT candidates with highest rank, several design disciplines were decoupled from the present analysis.

Seven major test phases P1 to P7 for accounting maximum DRE percentage all over STP fault injection and removal model (see Fig. 3) for DDT candidate selection in each test phase were determined. These were the Static Test Techniques – TT1 (consisting of three DDTs as *one factor* in optimization experiment applying Orthogonal Arrays as Optimization Strategy), the TT2 i.e. DDT_4= CEG+BOR+MI and TT3 – Hybrid Detection Technique= DDT_5 (consisting of Category Partition, Boundary value analysis, Path testing etc.). The objective of this investigation was then to determine the best combination of Test Techniques options for

the seven major test phase activities sections optimized for STD&STP maximum DRE percentage under cost and time constraints according to IOSTP benefit index maximization in (4) [25-27].

2. Design the experiment and select an appropriate orthogonal array

Owen [17], lists a set of orthogonal arrays for computer experiments. For this study, an orthogonal array that enables the study of seven variables with three levels each was selected (http://lib.stat.cmu.edu/designs/owen.small). If a full factorial design where all possible variable/TT combinations studied would have required 2,187 (3^7) experiments while in Orthogonal Array design of experiment plan only 18 experiments are enough as shown on Table 5. Variable interactions were assumed to be insignificant for this study.

Table 5 Seven Variable Orthogonal Array with three levels each [17]

No. of Exper.	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**
1	**1**	**1**	**3**	**2**	**3**	**1**	**1**
2	**3**	**2**	**2**	**1**	**1**	**3**	**1**
3	**2**	**3**	**1**	**3**	**2**	**2**	**1**
4	**1**	**2**	**1**	**1**	**2**	**1**	**2**
5	**3**	**3**	**3**	**3**	**3**	**3**	**2**
6	**2**	**1**	**2**	**2**	**1**	**2**	**2**
7	**1**	**3**	**2**	**1**	**3**	**2**	**3**
8	**3**	**1**	**1**	**3**	**1**	**1**	**3**
9	**2**	**2**	**3**	**2**	**2**	**3**	**3**
10	**1**	**3**	**1**	**2**	**1**	**3**	**1**
11	**3**	**1**	**3**	**1**	**2**	**2**	**1**
12	**2**	**2**	**2**	**3**	**3**	**1**	**1**
13	**1**	**1**	**2**	**3**	**2**	**3**	**2**
14	**3**	**2**	**1**	**2**	**3**	**2**	**2**
15	**2**	**3**	**3**	**1**	**1**	**1**	**2**
16	**1**	**2**	**3**	**3**	**1**	**2**	**3**
17	**3**	**3**	**2**	**2**	**2**	**1**	**3**
18	**2**	**1**	**1**	**1**	**3**	**3**	**3**

3. Conduct the orthogonal array experiments

The eighteen matrix experiments were conducted using a DRE estimating relationships in an post-mortem real project data doing "what-if" analysis i.e. which DRE percentage of all over IOSTP will be reached if we combine DDTs in different way par test phase activities (P1 to P7) according to Borda ranking result and Orthogonal Array design of experiment plan in 18 experiments from Table 5 where TT1 is codded as 1, TT2 as 2 and TT3 as 3. The analysis results of the 18 experiments for IOSTP DRE percentage and corresponding TT selection per each test phase are presented in Table 6. For the 18 DDT combinations shown in Table 6, the highest DRE is 94.44 % (experiment number seven).

Table 6 The "what-if" analysis results of OACE experiment

Exp. No.	P1	P2	P3	P4	P5	P6	P7	DRE (%)
1	1	1	3	2	3	1	1	90.51
2	3	2	2	1	1	3	1	84.79
3	2	3	1	3	2	2	1	87.66
4	1	2	1	1	2	1	2	91.27
5	3	3	3	3	3	3	2	80.34
6	2	1	2	2	1	2	2	81.66
7	**1**	**3**	**2**	**1**	**3**	**2**	**3**	**94.44**
8	3	1	1	3	1	1	3	83.14
9	2	2	3	2	2	3	3	82.99
10	1	3	1	2	1	3	1	87.89
11	3	1	3	1	2	2	1	85.05
12	2	2	2	3	3	1	1	89.77
13	1	1	2	3	2	3	2	83.91
14	3	2	1	2	3	2	2	85.19
15	2	3	3	1	1	1	2	81.72
16	1	2	3	3	1	3	3	84.11
17	3	3	2	2	2	2	3	82.58
18	2	1	1	1	3	1	3	92.94

4. Analyze the data to determine the optimum levels and verify results

The average DRE (%) for each variable P and for each of the three levels i.e. TT are calculated and displayed in the response table given in Table 7. This response table shows the DRE (%) effects of the variables at each level. These are separate effects of each parameter and are commonly called main effects. The average (%) shown in the response table are calculated by taking the average for a variable at a given level, every time it was used. As an example, the variable P1 was at level 2 in experiments 3,6,9,12,15 and 18. The average of corresponding DRE (%) is 86.12 (%) which is shown in the response table (Table 7) under P1 at level 2. This procedure is repeated and the response table is completed for all variables at each level.

Table 7 DRE (%) Response table per phase P

Phase / TT	P1	P2	P3	P4	P5	P6	P7
1	**<u>88.69</u>**	86.20	**<u>88.02</u>**	**<u>88.36</u>**	83.91	86.35	84.02
2	86.12	**<u>86.35</u>**	86.19	85.14	85.58	**<u>86.51</u>**	**<u>87.61</u>**
3	83.52	85.77	84.12	84.82	**88.72**	85.50	86.70

The optimum level (TT) for the design variables (P) can now be selected by choosing the level with the highest DRE percentage. For example the highest DRE percentage is got when variable P1 was at level 1 at 88.69 % as opposed to 83.52 % at level 3, and 86.12 % at level 2. Similarly, the levels that optimize total IOSTP defect removal effectiveness (DRE) were chosen. The optimum levels are indicated by bold&underlined in Table 7. As the next step, least squares regression analysis is used to fit the second order approximation model (Equation 10) to the DRE data (y_i) given in Table 6 in terms of the seven design variables (X_i). This parametric model accounts for the response surface curvature (square terms) and two factor interactions (cross terms) i.e. RSM:

$$DRE(\%) = 111.71 - 2.58(P1) + 1.22(P2) - 1.95(P3) - 7.61(P4) - 0.69(P5) + 0.94(P6) - 13.04(P7) - 0.36(P2)^2 + 1.46(P4)^2 + 0.79(P5)^2 - 0.36(P6)^2 + 3.15(P7)^2 \quad (11)$$

Note that, in this response surface approximation model, the parameter values are restricted to 1 (TTl), or 2 (TT2), or 3 (TT3).

Table 8 Maximum DRE (%) value and corresponding Test Techniques choices per test phase solution

P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**DRE (%)**
TT1	***TT2***	***TT1***	***TT1***	***TT3***	***TT2***	***TT2***	***94.03***

At these levels, the IOSTP DRE was predicted to be ***94.03*** % using a second order prediction model (10). As a next step, a verification analysis was performed. The DRE (%) of an IOSTP calculated from these test techniques choices, according to the post-mortem real project data using optimized DDT chices from Table 8, we computed DRE (%) to be 93.***43*** % . Difference is 0.6%=94.03%-93.43% that is acceptable to validate our prediction model for DRE (%) in equation (11) for optimal DDT combination choice given in Table 8.

Optimal combination of DDT choices per phase P given in Table 8 made increase of about 6 %, compared to un-optimized DDTs combination per each test phase we used in our real project in which we achieved DRE of 87.43 %.

5 Conclusion

Organizations are constantly working to leverage today's best practices for testing—within the context of their existing IT environments. As IT works to balance the business needs for a certain application and the testing limitations with regards to resources and schedules, making the best use of the testing environment becomes critical. Optimized testing is a way for organizations to move their testing efforts forward to reflect changing business environments and resource constraints. Optimized testing uses test techniques which has the highest defect detection yield and combined with the Orthogonal Array Testing Strategy (OATS) provides:

- Pairwise testing that protects against pairwise bugs while dramatically reducing the number of tests to perform which is especially cool because pairwise bugs represent the majority of combinatoric bugs and such bugs are a lot more likely to happen than the ones that only happen with more variables.
- Plus, the availability of tools means you no longer need to create these tests by hand.
- Pairwise testing might find some pairwise bugs while dramatically reducing the number of tests to perform, compared to testing all combinations because pairwise bugs represent the majority of combinatoric bugs.
- Plus, the availability of tools means you no longer need to create these tests by hand, except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results which improves application quality, maximizes development resources and helps deliver applications on time and within budget.

Using the tools associated with optimized testing, including DDT assessment, requirements management and automation, IT managers can make more informed decisions regarding testing, and have the data and information to back up those decisions.

This article has highlighted increase of DRE about 6 % compared to un-optimized DDTs combination per each test phase maximizing DRE percentage of STP solving optimization problem with best DDT choices combination in each phase P1 to P7 as controlled variables values which is determined by designed experiment plan using *orthogonal arrays* designed for this computer experiment (OACE).

References

[1] http://www.pairwise.org/tools.asp.

[2] P. E. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In Ninth Annual Conference on Computer Assurance (COMPASS'94), Gaithersburg MD, pages 69–80, 1994.
[3] J. Bach and P. Shroeder. Pairwise testing - a best practice that isn't. In Proceedings of the 22nd Pacific Northwest Software Quality Conference, pages 180–196, 2004.
[4] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and test coverage. In Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR), San Diego CA, 1998.
[5] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In Proceedings of the IEEE International Conference on Communications (Supercomm/ICC'94), May 1-5, New Orleans, Louisiana, pages 745–752, 1994.
[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions On Software Engineering, 23(7), 1997.
[7] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. IEEE Software, 13(5):83–87, 1996.
[8] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. In Proceedings of the IASTED International Conference on Software Engineering, 2004.
[9] S. R. Dalal, A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, and B. M. Horowitz. Model-based testing in practice. In Proceedings of the International Conference on Software Engineering (ICSE 99), New York, pages 285–294, 1999.
[10] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In Proceedings of the International Conference on Software Engineering (ICSE 97), New York, pages 205–215, 1997.
[11] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies - a survey. GMU Technical Report, 2004.
[12] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. Discrete Mathematics, 284(1-3):149–56, 2004.
[13] R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In Proc. of the 27th NASA/IEEE Softw. Engin.g Workshop, NASA Goddard Space Flight Center, 2002.
[14] Y. Lei and K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, pages 254–261, 1998.
[15] Paradkar, Amit: "*Specifcation Based Testing Using Cause-Efect Graphs*", *Ph.D. dissertation*, Graduate Faculty of North Carolina State University, COMPUTER SCIENCE, Raleigh, 1996.
[16] R. Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. Communications of the ACM, 28(10):1054–1058, 1985.
[17] Owen, A.: "Orthogonal Array Designs for Computer Experiments," Department of Statistics, Stanford University, 1994, http://lib.stat.cmu.edu/designs/owen.small.
[18] G. Sherwood. Effective testing of factor combinations In Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC, pages 133–166, 1994.
[19] H. Shimokawa and S. Satoh. Method of setting software test cases using the experimental design approach. In Proceedings of the Fourth Symposium on Quality Control in Software Production, Federation of Japanese Science and Technology, pages 1–8, 1984.
[20] B. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in testing the remote agent planner. In Proceedings of AIPS, 2000.
[21] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. IEEE Transactions of Software Engineering, 28(1), 2002.
[22] K. Tatsumi. Test case design support system. In Proceedings of the International Conference on Quality Control (ICQC), Tokyo, 1987, pages 615–620, 1987.
[23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. IEEE Trans. Software Engineering, 30(6):418–421, October 2004.
[24] A. W. Williams. Software components interactions testing:coverage measurement and generation of configurations, PhD thesis, Computer Science, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering,University of Ottawa, 2002.
[25] Lj. Lazić, The Integrated and Optimized Software Testing Process, PhD Thesis, School of Electrical Eng., Belgrade, Serbia, 2007.
[26] Lj. Lazić, Mastorakis, N. RBOSTP: Risk-based optimization of software testing process Part 2", *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS*, Issue 7, Volume 2, p 902-916, July 2005.
[27] Lj. Lazić, D. Velasević: "Applying simulation and design of experiments to the embedded software testing process", STVR, Volume 14, Issue 4, p257-282, John Willey & Sons, Ltd., 2004.
[28] T. Gregory: "The Economic Impacts of Inadequate Infrastructure for Software Testing", NIST Final Report, May 2002.
[29] S. H. Kan. *Metrics and Models in Software Quality Engineering*, Second Edition, Addison-Wesley, 2002.
[30] Brownlie R, Prowse J, and Phadke M.: " Robust Testing of AT&T PMX/StarMAIL Using OATS", *AT&T Technical Journal*; 71(3): 41- 47, May/June 1992.
[31] Pamela A. Engert, Zachary F. Lansdowne**.** *Risk Matrix User's Guide Version 2.2*, © 1999 The MITRE Corporation, 1999**.**