Techniques to reduce a set of test cases

Ljubomir Lazić, School of Computer Science, Knez Mihailova 6/VI, Beograd, SCG, llazc@raf.edu.yu
Nikos Mastorakis, Military Institutions of University Education, Terma Hatzikyriakou, 18539, Piraeus, Greece,

Abstract: The primary purpose of Software Testing Process and Evaluation (STP&E) is to reduce risk. All testing techniques provide insight and helps identify "unknown-unknowns". This paper describes STP with Assured Confidence techniques. The uncertainty of an important issue in the computer industry today and testing is still the main technique for quality assurance. There is a need to ensure that the software is reasonably safe from severe faults after testing. When faced with financial or a schedule constraints, testing is usually cut horizontally attempting to cover as many different test requirements at the expense of depth. We have reached a point where we must test smarter and apply Statistical-Risk-Based Test with Assured Confidence (SRBTAC) management procedure. We need to pick the right assessment tools to make vertical cuts in our test strategies. Any given test has a low probability of detecting a problem, but all the possibilities create a too-high probability of defects. To define the smallest number of tests you need to cover “enough” tests applying reduction hypotheses such as: uniformity, regularity, induction, deduction, analogy etc. and determine how many combinations you require for “enough” tests. If you analyze the inputs, there exists a set of combinations that defines the outputs, so you test ONLY those combinations of inputs. Approaches to software testing based on methods from the field of design of experiments have been advocated as a means of providing high coverage with minimal number of test cases at relatively low cost. These techniques can be easily embedded in an existing testing STP with minimal changes and extra effort.

Key-Words:- software testing, statistical testing, optimization, sampling, design of experiments.

1 Introduction
Testing is a critical component of modern software development. The problem of designing a suite of test cases is superficially similar to that of designing an experiment to estimate main effects and interactions, but there are crucial differences. Development and production of high-quality software at a reasonable cost is a critical issue for today’s products and services. Software testing is used to validate the correctness of programs as well as to weed out defects, and is typically the most expensive phase of the production process. Thus the problem of performing effective and economical testing is a key issue. Papers on this subject[1-18] appear in proceedings of several major conferences on testing (prominent ones in the US are International Conference on Testing, STAR conference, ICSE, and ISSRE), and in various IEEE and ACM journals. In testing we must generate test cases, administer them, correct faults when they are found, and manage the whole enterprise. Thus, there are (at least) three ways productivity can be improved, namely, generation of more efficient sets of test cases, automation of the testing process, and improvement of the management process. The last two issues are hard to influence since they require organizational changes and commitments [12,18]. Here we concentrate on the first issue, that of designing test cases. The purpose of this paper is to describe this approach, to survey known results [12-19] and present a few new ones, and to suggest some research opportunities. 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. To solve this problem, we revisit the reduction hypotheses [13], such as: uniformity, regularity, induction, deduction, analogy etc., combined with design of experiments (combinatorial and orthogonal vectors design) [15] for test generation and show that the size of the test suite which these techniques generates is few order of magnitude factor smaller in size than any other traditional test suite generation method or generate near optimal test suit size.

In section 2, the status of quality assurance in the software development process is analyzed, then in section 3 importance of sampling is discussed in context of the specification and oracles-based software testing: formal theory. The Statistical-Risk-Based Testing with Assured Confidence is presented in section 4 and application of Design of Experiments to reduce number of test cases in section 5 is described. Finally, in section 6 main results and future research is concluded.

2 The status of quality assurance in the software development process
Quality assurance measures, such as testing during the software development process, are recognized software engineering methods and are included in every software process model. In practice it is important to design in software product quality in each software development
step, from beginning to the product delivery and maintenance phase. Software development is based on four conflicting dimensions: functionality, quality, deadlines and costs. The scope for solutions on any given project is restricted with regard to these dimensions. Thus, if the costs and deadline are specified in advance, the functionality and/or quality assurance have/has to be limited. So, quality assurance is restricted by the dimension of costs on the one hand and time-to-market requirements on the other. For software producers, quality assurance especially in the form of software testing entails effort which, in turn, affects the release date for the software product. This is why it is often doubted that the effort invested in quality assurance is necessary. However, the economic viability of quality assurance has to be measured using other criteria, e.g. at the user level, where the software is in productive use. There is no doubt that system crashes and malfunctions cost more effort on the part of the user. Unfortunately, there are hardly any representative studies on this issue [2,6,16-19]. Testing is expensive; industry data indicates that about half the software budget is spent on testing. Testing costs are best attacked in the development process, by clarifying and simplifying requirements, providing for testability and test automation, and verifying code against specifications. When high quality software reaches the test organization, there are two goals: (1) provide the development organization with the most useful information possible as quickly as possible in order to shorten the overall development cycle, and (2) certify the system as quickly and inexpensively as possible. Just “more testing” will certainly add cost, but will not necessarily add new information or significantly improve reliability estimates [12-19]. When faced with financial or a schedule constraints, testing is usually cut horizontally attempting to cover as many different test requirements at the expense of depth. We have reached a point where we must test smarter and apply Statistical-Risk-Based Test with Assured Confidence (SRBTAC) management procedure [19]. We need to pick the right assessment tools to make vertical cuts in our test strategies. Defense systems are becoming increasingly software intensive. While software enhances the effectiveness and flexibility of these systems, it also introduces vulnerabilities related to inadequacies in software design, maintenance, and configuration control. Effective testing of these systems must take into account the special vulnerabilities introduced by software. The software testing problem is complex because of the astronomical number of scenarios and states of use. The domain of testing is large and complex beyond human intuition. Because the software testing problem is so complex, statistical principles must be used to guide testing strategy in order to get the best information for the resources invested in testing. In general, the concept of “testing in” quality is costly and ineffectual; software quality is achieved in the requirements, architecture, specification, and design and coding activities. The problem of doing just enough testing to remove uncertainty regarding critical performance issues and to support the decisions that must be made in the software life cycle is a problem amenable to solution by statistical science. In order to significantly improve software testing efficiency and effectiveness for the detection and removal of requirements and design defects in our framework of the Integrated and Optimized Software Testing Process (IOSTP) [14,15]. During 3 years of our 6σ deployment to STP we calculated overall value returned on each dollar invested i.e. ROI of 100:1 [12]. Unlike conventional approaches to software testing (e.g. structural and functional testing) which are applied to the software under test without an explicit optimization goal, the IOSTP framework [14] with embedded Risk Based Optimized STP (RBOSTP) approach designs an optimal testing strategy to achieve an explicit optimization goal, given a priori [16,17]. This leads to an adaptive software testing strategy. A non-adaptive software testing strategy specifies what test suite or what next test case should be generated, e.g. random testing methods, whereas an adaptive software testing strategy specifies what testing policy should be employed next and thus, in turn, what test suite or test case should be generated next in accordance with the new testing policy to maximize test activity efficacy and efficiency subject to time-schedule and budget constraints. The process is based on a foundation of operations research, experimental design, mathematical optimization, statistical analyses, as well as validation, verification, and accreditation techniques. The use of state-of-the-art methods and tools for planning, information, management, design, cost trade-off analysis, and modeling and simulation, Six Sigma strategy significantly improves STP effectiveness. Figure 1 graphically illustrates a generic IOSTP framework [14].

![Image](Fig. 1 Integrated and optimized software testing process (IOSTP) framework [14])
2.1 Importance of sampling

Our testing approach is based on the concepts of software testing based on specifications, relies on a solid theoretical framework of set of reduction hypotheses $H_R$ applicable to the program $P$ [oracle], software documentation, statistical principles and Design of Experiments in order to reduce the size of a test set or even find minimal test suite size. We apply reduction hypotheses to the program in order to avoid redundancy in test case selection step; also we assume certain knowledge of the behavior of the program that is not necessary to test. This reflects common test practices. There are two basic ways of showing that code is 100% correct: (1) program proving and (2) exhaustive testing. Neither technique, as yet, is an option when it comes to major real-life systems. Techniques for proving software correct are not mature enough, while exhaustive testing is not practical because the costs are prohibitive. To reduce the exhaustive testing process to a feasible testing process, we must find criteria for choosing representative test-cases that are comprehensive enough to span the problem domain and provide us with the required degree of confidence that the software works correctly and reliably. At the same time, this test set must be small enough so that the testing can be performed within the available schedule and cost constraints. Testing is considered complete or adequate under a certain metric, if, on execution, test cases exercise (cover) at least once a pre-determined fraction of the metric structures that exist in the program. The statistical testing approach to software treats the software like a statistical experiment. A statistical subset of all possible software uses is first generated. Performance on this subset is used to form conclusions about operational performance based on the usage model developed. The expected operational use is represented in a usage model of the software. Test cases are then randomly generated from the usage model. These tests are executed in an operational environment. Most usage modeling and related statistical testing experience to date is with embedded real-time systems, application program interfaces, and graphical user interfaces. One very advanced industrial user of this technology is in the mass storage devices business. Use of this technology has led to extensive test automation, improved feedback to the developers regarding product deficiencies or quality, improved advice to management regarding suitability for deployment, and greatly improved field reliability of products shipped. From a statistical point of view, all the topics in this paper follow sound problem solving principles and are direct applications of well established theory and methodology. From a software testing point of view, the application of statistical science is relatively new and rapidly evolving, as an increasing range of statistical principles is applied to a growing variety of systems. Statistical testing is used in pockets of industry and agencies of government, including the DoD, on both experimental and routine bases. Since exhaustive testing is out of the question, sampling criteria are an essential part of the testing approach [13]. In next section, we present the main reduction hypothesis for sampling criteria. This paper will focus on the successes and issues associated with developing a statistical testing environment for an industrial software project. The paper will also describe how both statistical testing based on software models and traditional testing based on unit and other functional tests can be combined into an effective approach to testing large software intensive systems.

3 The specification and oracles-based software testing: formal theory

Our approach to testing software from formal specifications relies on a solid theoretical framework, software documentation, modeling & simulation. It is an adaptation and generalization of the approach reported by G. Bernot, M. C. Gaudel, and B. Marré i.e. BGM method [20] and an adaptation to object-oriented systems of the BGM method presented in [6,8]. The BGM method has been developed for testing data types by using formal specifications. However, this method is oriented towards algebraic specifications, and does not fulfill the needs of object-oriented software development. The formal testing method is an approach to detect errors in a program by validating its functionalities without analyzing the details of its code, but by comparing it against a specification. The goal is to answer the question: "Does a program satisfy its formal specification?" or, in accordance with the goal of testing, to find out if a program does not satisfy its specification. The formal testing process is usually decomposed into the following three phases as depicted in figure 2:

- **Phase 1 Test selection:** the test cases that express the properties of the specification are generated.
- **Phase 2 Test execution:** the test cases are executed and results of the execution collected.
- **Phase 3 Test satisfaction:** the results obtained during the test execution phase are compared to the expected results. This last phase is performed with the help of an oracle. Our oracle is a mechanism based on external observation of the behavior of the tested program. This section presents the whole test process of the formal testing theory, starting from the test foundation and then focusing on the test selection and test satisfaction phases i.e. oracles-based testing showed in figure 3. Throughout this section we use the following notations:
  - $SPEC$: class of all specifications written in the specification language considered,
  - $PROG$: class of all programs expressed in the language used for the implementation,
  - $TEST$: class of all test sets that can be written,
• |= : satisfaction relationship on \( PROG \times SPEC \), expressing the validity of the program with respect to the specification,
• \( |=_O \) : satisfaction relationship on \( PROG \times TEST \), deciding if the test cases are successful or not for the program under test. \( |=_O \) is the oracle satisfaction relationship.

However, the oracle \( |=_O \) can only be constructed for a program \( P \) and a test set \( T_{SP} \) if \( T_{SP} \) is practicable, i.e.:
• \( T_{SP} \) is practicable, i.e. it has a “reasonable” finite size, which is not the case of the exhaustive test set for arbitrary specifications, and
• \( T_{SP} \) is decidable, i.e. it is possible to decide whether the program is acceptable or not for the given specification.

\( T_{SP,H} \) is the practicable test set as a result of sampling criteria i.e. test selection is a function of the specification and of the reduction hypotheses \( H_R \), TC – testing constraints (time, budget, cost) and \( \alpha \) – acceptable risk (uncertainty level).

Figure 2. Specification-based testing

3.1 Test selection

Assuming that we have an oracle \( O \) that ensures the observability of the system with the oracle hypotheses \( H_O \), the first task of the test procedure consists of selecting, from the specification, a test set that allows the exhaustive validation of each service required from the system. This is theoretically achieved by selecting an exhaustive test set which contains the entire test that is required by the specification. However the exhaustive test set selected is generally infinite, and it is necessary to apply a number of reduction hypotheses \( H_R \) to the behavior of the program to obtain a finite test set of “reasonable” size. Note that the test procedure itself is successful if the test set helped finding as many errors as possible, i.e. if the test set is unsuccessful. The efficiency of test set, i.e. its aptitude (capability) to detect errors, must be aligned to some quality criteria that depend on time and budget constraints. Therefore, we proceed by successive reductions of the number of tests (see Figure 4.). Thus, when the test set is successful, the program is correct on condition that it satisfies the oracle and the reduction hypotheses. The test set quality is therefore function of the number of oracle and reduction hypotheses satisfied by the program under test.

3.2 Test satisfaction

Once a test set has been selected, it is used during the execution of the program under test. Then the results collected from this execution must be analyzed. For this purpose, it is necessary to have a decision procedure to verify that an implementation satisfies a test set. This process is called oracle-based test process model as depicted in Figure 3. The oracle is a program that must decide the success or failure of every test case, i.e. whether the evaluation of test cases is satisfied or if test cases reveal errors.

![Figure 3. Testing model with Oracle](image)

**Definition 3. Oracle**

The oracle \( O = (|=_O, Dom_O) \) is a partial decision predicate of a formula in a program \( P \in PROG \). For each test case \( t \in TEST \) belonging to the oracle domain \( Dom_O \), the satisfaction relationship \( |=_O \) on \( PROG \times TEST \) allows the oracle to decide:
• If \( t \) is successful in \( P (P |=_O t) \).
• If the answer is inconclusive (\( t \) is non-observable). ⋄

The oracle is constituted of a collection of equivalence relationships that compare similar elements of the scenario derived from the specification to the program under test; these elements are said to be observable. The problem is that the oracle is not always able to compare all the necessary elements to determine the success or failure of a test; these elements are said to be non-observable. This problem is solved using the oracle hypotheses \( H_O \) which are part of the possible hypotheses.
and collect all power limiting constraints imposed by the realization of the oracle:

**Definition 2. Oracle Hypotheses**

The oracle hypotheses \( H_0 \) are defined as follows:

- When a test case \( t \in \text{TEST} \) is observable (\( t \in \text{Dom}_o \)) for a program \( P \), the oracle knows how to decide the success or failure of \( t \): 
  \[
  (P \text{ satisfies } H_0) \Rightarrow ((P \models_o t) \lor (\neg (P \models_o \neg t))).
  \]

- When a test case \( t \) is non-observable for a program \( P (t \in \text{Dom}_o) \), the oracle has a set \( C \) of criteria \( c_i \) allowing to observe the outcome of \( t 
  \[
  (P \text{ satisfy } H_0 \land P \models_o (c_i \in C \land c_i (t))) \Rightarrow (P \models_o t). \]

The first hypothesis stipulates that for any observable test case, the oracle is able to determine whether the test execution yields yes or no, i.e. that no test case execution remains inconclusive. The second hypothesis stipulates that for any non-observable outcome of test case, there are criteria to transform it into an observable outcome of test case. Since the oracle cannot handle all possible transformers that are proposed as test cases, oracle hypotheses must be taken into account to limit the test selection to decidable test transformers. Thus, it seems rational to put the oracle hypotheses \( H_0 \) at the beginning of the test selection phase of the test process. Moreover, transformer can lead to non-deterministic behaviors. Thus, another necessary oracle hypothesis is the assumption that this non-determinism is bounded and fair. In this way, non-deterministic mechanisms can be tested by a limited number of applications of the same test case. Several observations can be made from the model of program behavior transform. Different types of oracles are needed for different types of software and environments. The domain, range, and form of input and output data varies substantially between programs. Most software has multiple forms of inputs and results so several oracles may be needed for a single software program. For example, a program’s direct results may include computed functions, screen navigation, and asynchronous event handling. Several oracles may need to work together to model the interactions of common input values. If we consider a word processor, pagination changes are based upon the data being displayed and characteristics such as the page width, point size, page layout, and font. In Windows, the current printer driver also affects the pagination even when nothing is printed. Just changing the selected printer to one with a different driver can change the pagination of a Word document. Although an oracle may be excellent at predicting certain results, only the SUT running in the target environment will process all of the inputs and provide all of the results. 

### 3.3 Reduction hypotheses for test selection

In order to reduce the size of a test set or even find minimal test suite size, we apply reduction hypotheses to the program in order to avoid redundancy in test case selection step; also we assume certain knowledge of the behavior of the program that is not necessary to test. This reflects common test practices. There are two basic ways of showing that code is 100% correct-program proving and exhaustive testing. Neither technique, as yet, is an option when it comes to major real-life systems. Techniques for proving software correct are not mature enough, while exhaustive testing is not practical because the costs are prohibitive. To reduce the exhaustive testing process to a feasible testing process, we must find criteria for choosing representative test-cases that are comprehensive enough to span the problem domain and provide us with the required degree of confidence that the software works correctly and reliably.

To simplify, \((H, T_{SP, H})_0 \) is noted \((H, T)_0 \) in the rest of the article. The selection of a pertinent test set \( T \) of “acceptable” size is performed by successive refinements of an initial test context \((H^0, T^0)_0\), which has a pertinent test set \( T^0 \) of unreasonable size, until the obtaining of a practicable test context \((H, T)_0\). This refinement of the context \((H^i, T^i)_0\) into \((H^j, T^j)_0\) induces a pre-order between contexts:

\[
(H^i, T^i)_0 \leq (H^j, T^j)_0
\]

At each step, the pre-order refinement context \((H^i, T^i)_0 \leq (H^j, T^j)_0 \) is such that:

- The hypotheses \( H^j \) are stronger than the hypotheses \( H^i \):
  \[
  H^i \Rightarrow H^j.
  \]
- The test set \( T^i_{SP, H_j} \) is included in the test set \( T^i_{SP, H_j} \):
  \[
  T^i_{SP, H_j} \subseteq T^i_{SP, H_j}.
  \]
- If \( P \) satisfies \( H^j \) then \((H^i, T^i)_0 \) does not detect more errors than \((H^j, T^j)_0\):
  \[
  (P \text{ satisfies } H^j) \Rightarrow (P \models_o T^i_{SP, H_j} \Rightarrow P \models_o T^j_{SP, H_j}).
  \]
- If \( P \) satisfies \( H^j \) then \((H^i, T^i)_0 \) detects at least as many errors as \((H^j, T^j)_0\):
  \[
  (P \text{ satisfies } H^j) \Rightarrow (P \models_o T^i_{SP, H_j} \Rightarrow P \models_o T^i_{SP, H_j}).
  \]

The strength (weak or strong) of a reduction hypothesis is linked to the probability that the program satisfies the hypothesis: a hypothesis with a high probability of satisfaction by the program is weak whereas a hypothesis with a low probability of satisfaction by the program is strong. At the same time, this test set must be small enough so that the testing can be performed within the available schedule and cost constraints. Testing is considered complete or adequate under a certain metric, if, on execution, test cases exercise (cover) at least once a pre-determined fraction of the metric structures that exist in the program. For example, 90% branch coverage criterion requires that at least 90% of all the branches in the program be executed at least once by some test case. Structural and program-based strategies include statement and branch testing, data-flow testing, domain testing, mutation testing and other [5]. However, there are a number of practical problems with this category of methodologies as well.
The uniformity hypotheses help to limit the size of the test set by making the assumption that if a test is successful for all terms whatever their complexity, then it is successful for all terms having a complexity less than or equal to a defined bound, then it should behave correctly for all possible instantiations i.e. if a transformer \( TR \), for example is formula \( f \), containing a term \( t \), is successful for all terms \( t \) which have a complexity less than or equal to a bound \( k \), then it is successful for all possible complexities of \( t \). The regularity hypotheses help to limit the size of the test set by making the assumption that if a test is successful for terms having a complexity less than or equal to a given bound, then it is successful for all terms whatever their complexity.

The strength (weak or strong) of a reduction hypothesis is linked to the probability that the program satisfies the
hypothesis: a hypothesis with a high probability of satisfaction by the program is weak whereas a hypothesis with a low probability of satisfaction by the program is strong. In the present state of the art, we do not dispose of a measure of the strength of the hypotheses. However, in most cases, test sets selected by uniformity are included in test sets selected by regularity, because the generalization $m:n$ is likely to include the case of the generalization $1:n$. That is why we consider the uniformity hypotheses to be stronger than the regularity hypotheses, even though this is not an absolute rule.

For instance, in the case of an account, a measure of complexity is the number of transactions. We could imagine a regularity hypothesis stipulating that if an object reacted correctly to 20 transactions, then it will react correctly to any number of transactions.

4 Statistical-Risk-Based Testing with Assured Confidence

Statistical testing can be initiated at any point in the life cycle of a system, and all of the work products developed along the way become valuable assets that may be used throughout the life of the system.

An operational usage model is a formal statistical representation of all possible uses of a system. A usage model may be represented in the familiar form of a state transition graph, where the nodes represent states of system use and the arcs represent possible transitions between states. If the graph has any loops or cycles (as is usually the case), then there is an infinite number of finite sequences through the model, thus an infinite population of usage scenarios. In such graphical form, usage models are easily understood by customers and users, who may participate in model development and validation. As a statistical formalism, a usage model lends itself to statistical analysis that yields quantitative information about system properties. The basic task in model building [9] is to identify the states-of-use of the system and the possible transitions among states-of-use. Every possible scenario of use, at the chosen level of abstraction, must be represented by the model. Thus, every possible scenario of use is represented in the analysis, traceable on the model, and potentially generated from the model as a test case. The logical conclusion is that simpler tests allow you meet overall test goals sooner with a higher probability of success. Statistical-Risk-Based Test with Assured Confidence (SRBTAC) management procedure can be run by these steps:

1. Define all system requirements (potentially to be tested)
2. Identify Risk Assessment techniques
3. Identify high risk requirements
   a. Identify consequence of faults for each requirement
   b. Identify fault probability indicators (if possible)
4. Plan and define tests according to requirement prioritisation and coverage criteria as set out in test plan
5. Execute test according to prioritisation and acceptance criteria as defined in the test plan
6. Collect metrics to monitor progress and report on priority level coverage (i.e. How many of the requirements per priority level have been tested)
7. Repeat until acceptance criteria per priority level has been met (i.e. Number of outstanding faults for each priority level is acceptable)

This section outlines two processes for conducting the SRBTAC management procedure as described in [19]. The first is a bottom-up process where the SRBTAC can be used during software development. The second is a top-down process that can be applied to projects where testing has been almost completed. The top-down process has been used at several testing sites with encouraging results. Before the process starts, one must determine the threshold failure density and the corresponding confidence level. The threshold failure density must be determined by the requirements only. The number of daily transactions and criticality of failures determine the threshold failure density. For example, if the system is mission-critical and no failure can be tolerated, the threshold should be low, say 0.0001. Once the threshold failure density is determined, the confidence level can be determined, and this again can be determined by the requirements. Note that higher confidence level and lower threshold on failure density increase the number of test cases needed.

4.1 Test Case Selection

The statistical model requires that test cases for the SRBTAC must be selected randomly and independently. However, completely random test cases may not cover all the important partitions in the input domain. Thus, this paper recommends that the input domains are thoroughly analyzed to identify major partitions, and then test cases are generated from these partitions to ensure coverage. Major partitions can be identified by examining the constraints on inputs, outputs and major execution paths in the code or design. Avoid any apparently dependent test cases like the following. Even though the above two cases are different, they drive the software with the same functionality and hence dependent. At each level of SRBTAC testing, only test cases from that level can be used. For example, at the SRBTAC end-to-end testing, only end-to-end test cases can be counted, but not integration or module test cases. Most engineers have good ideas where in software faults reside, and thus it is advantages to ask those engineers to prioritize those areas in their testing process. The testing team should emphasize on reuse of testing resources and
effort. In addition to regression testing, new test cases can be developed by composing and reusing existing test cases. During integration testing and end-to-end testing, the operating environment should be considered, and this may include external systems interfacing, physical environment, input data, system operators and end users. For each factor, identify those that are SRBTAC related and identify contingency plans for environment components that can not be certified.

**Determine the Confidence Level**
- Determine the confidence level required for a given system before conducting the SRBTAC process.
- For mission-critical system this can be 0.95, otherwise a lower confidence level, such as 0.8.
- The higher the confidence level, the more test cases are required.
- Conflict between the desired confidence level vs. the extra effort needed.

**Determine the Target Failure Rate**
- Determine the failure rate from the system requirements.
- If the system is mission-critical and safety-critical, use a low failure rate.
- The lower the failure rate, the more the number of test cases needed.
- The following figures show the number of test cases required.

**Test Cases Generation and Selection**
- Test cases must be randomly generated/selected for the SRBTAC.
- The test cases used must be independent of each other.
- The number of test cases must be large enough to achieve the desired confidence level.
- Randomness and Independence of test cases are two important aspects for consideration.

**Generate Independent Test Cases**
- Truly random selection or generation of test cases may not be the best approach, because important partitions may be lost.
- Test cases should cover major partitions
  - List all inputs and constraints on these inputs
  - Identify the major partitions of the inputs and constraints
- Partition can be identified based on input, output and execution paths
  - Randomly generate test cases in the major partitions
  - Avoid apparently dependent test cases, even though these the cases are different, but they drive the software with the same functionality.
- Random constraints
  - Some test case execution sequences must direct the system to a desired state.
  - Not all the parameters of a test cases can be randomly generated each time.

- Boundary and partition constraints can be considered to make sure the desire output state of the test case can be predicted.

**SRBTAC Statistical Models**
In article [19] we described two Statistical Model with corresponding equation SRBTAC I and II which are used to determine required number of test cases to assure confidence to detect defect in software. Equation SRBTAC I is used to calculate the number of test case required to achieve a certain level of confidence C that the failure density is no more than a desired bound B according to Table 1. All N test cases must execute correctly without causing the software to fail.

<table>
<thead>
<tr>
<th>Confidence Level(C)</th>
<th>Failure Density(B)</th>
<th>Number of test cases required</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>0.01</td>
<td>100</td>
</tr>
<tr>
<td>0.8</td>
<td>0.02</td>
<td>80</td>
</tr>
<tr>
<td>0.8</td>
<td>0.05</td>
<td>31</td>
</tr>
<tr>
<td>0.8</td>
<td>0.10</td>
<td>19</td>
</tr>
<tr>
<td>0.95</td>
<td>0.01</td>
<td>298</td>
</tr>
<tr>
<td>0.95</td>
<td>0.02</td>
<td>148</td>
</tr>
<tr>
<td>0.95</td>
<td>0.05</td>
<td>58</td>
</tr>
<tr>
<td>0.95</td>
<td>0.10</td>
<td>28</td>
</tr>
</tbody>
</table>

The equation SRBTAC I is applicable when every test case is successful. If one or more test cases fail, equation SRBTAC II should be used. During the SRBTAC experiments at several testing applications, it is apparent that some of the testing projects have some failures, but it is still necessary to compute the confidence. If N random test cases are executed with Q failures, one has the confidence C that the true failure rate is no more than B. In other words, with a probability of at least C, one will see more than Q failures in N test cases when the failure density is more than B. This model has the following characteristics:

1. Confidence value is between 0 and 1.
2. The maximum confidence from a given set of N test cases is obtained when there are no failures. By substituting Q = 0, one can obtain the SRBTAC I equation.
3. As the failure increases, the confidence decreases rapidly. When all test cases result in failures, the confidence is zero.
4. As the targeted failure density decreases, more test cases or fewer failures are required to achieve the same confidence.

Figure 1 shows how the confidence varies with the number of test cases for the target failure density 0.05, for the failures between Q = 0 and 5. Note that as the failures increases, the confidence decreases rapidly which is evident from the graphs becoming closer to the x-axis. When the failures increases from 0 to 5 out of 100 test cases, the confidence drops from 0.99 to
around 0.4. In figure 2, for a fixed bound \( B = 0.1 \) the confidence is a function of failures detected for various number of test cases. For a 0.1 target failure rate with 0.95 confidence, the failures should be less than 5 out of 100 test cases. If the targeted confidence level is 0.8, the failures should be less than 7 out of 100 test cases. Note that reduction in the confidence does not increase the failures significantly.

**Figure 1: Confidence in the Presence of Failures**

This is to be expected because the confidence decreases rapidly with each additional failure. Hence, in practice only few failures can be tolerated. Another important issue in using this new model is that the failures detected should not be critical. A single critical failure can disable mission-critical applications. Failures detected must be handled and tested after the SRBTAC process.

**Figure 2: Change in Confidence with Increasing Failures**

**Benefits and Experience of SRBTAC**

- It is relatively easy to identify which parts of subsystems are over tested and which are under tested. The testing team indicated that they can easily incorporate the SRBTAC requirements in their test projects if they were informed at the beginning of the project.
- The testing team indicated that it is easy to apply the SRBTAC process after some training.

**5 Design of Experiments in software testing (DOE)**

The process of software testing is typically divided into various phases: Unit testing (testing of small pieces of code written typically by one programmer), Integration testing (testing of several subsystems, each of which is comprised of many units) and System testing (testing of combination of subsystems). There may be still further phases such as Acceptance testing (when the software is first delivered to a client), Alpha testing (unofficial trials by willing customers before full product utilization), FOA (First Office Application), etc. Besides these stages of testing, there are many different methods of testing. Structural testing or, White box testing, refers to the type of testing in which tests are designed on the basis of detailed architectural knowledge of the software under test. Functional testing, or Black Box testing, refers to the type of testing where only the knowledge of the functionality of the software is used for testing; knowledge of the detailed architectural structure, or of the procedures used in coding, is not utilized. Structural testing is typically used during unit testing, where the tester (usually the developer who created the code) knows the internal structure and tries to exercise it based on detailed knowledge of the code.

Functional testing is used during integration and system test, where the emphasis is on the user’s perspective and not on the internal workings of the software. Functional testing tries to test the functionality of the software as it is perceived by the end users (based on user manuals) and the requirements writers. Thus, functional testing consists of subjecting the system under test to various user controlled inputs, and watching its performance and behavior. The primary focus of this paper is on functional testing. Since the number of possible inputs is typically very large, testers need to select a sample, commonly called a *suite*, of test cases, based on effectiveness and adequacy. Much functional testing is done in an intuitive and less formal manner. Typically, testers, working from their knowledge of the system under test and of the prospective users, decide on a set of specific inputs. Clearly there is the possibility that important interactions among the inputs will be missed. Herein lie significant opportunities for a systematic approach, based on ideas from sampling and experimental design theory. Consider the following example. In a stochastic simulation like in our case study of ATTRS [15] you’d really like to know all about the output variables distributions. So you usually have to settle for various summary measures of the output distributions. Traditionally, people have focused on estimating the *expected value* (or mean) of the output variable distribution and this can be of great interest. For people without statistical training, it can be difficult to organize information about the system under study in a way that aids the design of the experiment. To help clarify this process, we break the design task itself into five separate steps.

1. Define the goals of the experiment.
2. Identify and classify independent and dependent variables.
3. Choose a probability model for the behavior of the simulation model.
4. Choose an experiment design plan.
5. Validate the properties of the chosen design.

After the goals definition, appropriate DOE plan should be chosen. One determines the number of distinct model settings to be run and the specific values of the factors for each of these runs. There are many strategies for selecting the number of runs and the factor settings for each run. These include random designs, optimal designs (one of them is chosen in ATTRS case study in [15]), combinatorial designs, mixture designs, sequential designs, and factorial designs. Here we want to emphasize one method of experimental design applied to software testing i.e. the Orthogonal Array-based Robust Testing [1-4,7-9], based on Taguchi Robust Design which has a mathematical foundation in linear algebra—specifically, the Galois field theory—began with Euler as Latin squares, which is exploited in ATRRS field testing (see [15] section 5). Black-box testing of software components and system is indispensable and requires test input data for all input parameters. The number of test cases needed for exhaustive testing i.e. for all possible combinations of input data is usually extremely large — almost always too large for allocated testing resources that are always limited. The only solution is intelligent test case generation to cut down costs and improve the quality of testing. The Orthogonal Array-Based Robust Testing method has been used to test software from many diverse industries, e.g., telecommunications, automotive, and electromechanical systems. The users have typically reported a factor of 2-to-1 or better productivity improvement compared to their traditional testing methods [1]. The number of tests needed for this method is similar to the number of tests needed for the one-factor-at-a-time method, and with a proper software tool, the effort to generate the test plan can be small. Its ability to find faults is much better than one-factor-at-a-time method and approaches 100 percent, especially when used in conjunction with code coverage analysis tools. The test cases generated by this method have the highest effectiveness, measured in terms of the number of faults detected per test. Let us set forth a simple scenario for software testing. Consider a system under test with 4 components (factors designated as A, B, C, D respectively) which has 3 possible elements (levels designated as 1, 2, 3):

<table>
<thead>
<tr>
<th>Calling phone (factor A)</th>
<th>Call type (factor B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular phone</td>
<td>Local call</td>
</tr>
<tr>
<td>Mobile phone</td>
<td>Long distance</td>
</tr>
<tr>
<td>Coin phone</td>
<td>Toll free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access (factor C)</th>
<th>Called phone (factor D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISDN</td>
<td>Regular phone</td>
</tr>
<tr>
<td>PBX</td>
<td>Mobile phone</td>
</tr>
</tbody>
</table>

Exhaustive testing suite would require the set of test cases to be executed for $3^n = 81$ possible configurations for changing one-factor-at-a-time method. The OART algorithm L9 calculates only 9 combinations that are already sufficient to cover all pair-wise component interactions showed in Table 1. The rows correspond to test cases; the columns correspond to the test parameters. Thus, the first test case comprises Level 1 for each parameter, i.e., it represents the combination A1, B1, C1, D1. The second test case comprises combination A1, B2, C2, D2, etc. An orthogonal array (OA) has the balancing property that, for each pair of columns, all parameter-level combinations occur an equal number of times. In OA L9, there are nine parameter-level combinations for each pair of columns, and each combination occurs once.

By applying the algorithm based on Orthogonal Arrays and Covering Arrays, the overall number of test cases can be dramatically reduced compared to exhaustive testing with certainty that test coverage of all pair-wise input parameter or software components (i.e. configurations) combinations is achieved [26] such as in next example. Suppose that another SUT has 13 input parameters each with 3 input parameters domains. Then exhaustive testing would require $3^{13} = 1.594.323$ possible input parameter combinations. The OART algorithm calculates only 15 set of all pair-wise configurations that are already sufficient to cover all pair-wise input parameter interactions. The OART is applicable to unit test for test input data derivation for black-box testing methods, for integration and system scenario black-box test method, for configuration testing, for interoperability testing as well as for web testing.

The main outstanding technical problem is that of constructing efficient covering designs in an unified manner. It is not clear that combinatorial complexity theory has anything to say about how difficult this problem is. The approach we chose for this paper uses formally designed experiments, called screening designs, that are highly economical and whose primary purpose is to identify important low-order effects, i.e., first-, second-, or third order effects, where an n$^\text{th}$-order effect is an effect caused by the simultaneous interaction of n factors. For instance, for certain web server applications, a 1$^\text{st}$-order effect might be that performance slows considerably when logging is turned on and another might be that it also slows when few server threads are used. A 2$^\text{nd}$ order effect involves the interaction of two options, e.g., web server performance may slow when caching is turned off and the server performs blocking reads.

In our case study [15], testing of an Automated Target Tracking Radar System (ATTRS) is described. At the beginning of ATTRS development it was known only requirement specification for automated target tracking quality described in natural language as:
**Specification:** Maximum mean and standard deviation error ($\mu, \sigma$) of estimated target range from radar position should be less than 100 m and for azimuth of target position should be less than 1° with confidence level $\beta=0.95$ under conditions: target speed in range 0-300 m/s, radial acceleration of target maneuver in range 0-6g, radar target detection probability $P_d \geq 0.8$, relative error in ($\mu, \sigma$) estimation should be $r \leq 10\%$.

Design and test engineers worked together in order to exchange application domain knowledge. They identified that target trajectory can be modeled by combination of straight line, circle and parabolic curve. During planning phase we identified that accuracy of extractor in estimating target echo center position ($\mu, \sigma$ of R and $\theta$) has high importance because extractor’s output is input to Kalman tracking filter. High importance is maneuver detector (K- constant) in output is input to Kalman tracking filter. High radar target detection probability $P_d$. We needed information how this parameters impact to automatic target tracking quality. To test this software system, combinations of all these inputs must be provided and the output from the software system checked against the corresponding physics. Each combination tested is called a test case. One would like to generate test cases that include inputs over a broad range of permissible values. Since in this example, we have continuous variables as inputs, the total number of possible test cases is unlimited. To reduce the number of test cases, testers have developed a number of heuristic strategies. Two guiding principles are: 1) non-redundancy (test cases are chosen so that for each test case that is included, the number of test cases which remain to be tried is reduced by at least one), and 2) generality (the outcome of the test case is generalizable beyond the specific inputs used). To implement these principles a number of concepts have been developed which can be applied in conjunction with each other. One of these relates to the notion of Equivalence Partitioning. It is assumed that the range of each of the input variables can be divided into a number of mutually exclusive classes, called equivalence classes, with the property, that the outcome of a test is generalizable to the entire equivalence class. That is, the same outcome would be expected regardless of a specific input value from that class. One can reasonably assume that a test of a representative value of each class is equivalent to a test of any other value. Since one cannot “reasonably assume” unless there is only one member in an equivalence class, in practice testers divide the input domain into a number of possibly overlapping classes (but usually with very little overlap) and select from 1 to 3 distinct inputs as representatives from each class. Typically there is much freedom in choosing the partitioning. Having formed the equivalence partitioning, one still needs to decide which members be considered as representative members. This is where another notion, that of Boundary Value Analysis is applied. This is based on the experience that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary conditions are described as those situations directly on, or above, and beneath the edges of input equivalence classes. Thus, this concept is similar to that of a minimax strategy. Let us illustrate these concepts in context of the example we discussed above.

The system test team took the approach of developing a family of simulations, physical experiments (in laboratory and field facilities) to preict and confirm system performance using optimized test scenario based on design of experiment to minimize number of tests described in section 2. Because precision of automatic target tracking i.e. $\mu, \sigma$ error of estimated target range and azimuth from radar position with confidence level $\beta=0.95$, is statistical in nature we estimated number of experiment replication in order to assess ($\mu, \sigma$) with required precision $r \leq 10\%$ as described in [15] using well known equations from statistics we can assess lower limit $\sigma_1$ for $\sigma$ with confidence level 1-$\alpha_2$, and higher limit $\sigma_2$ for $\sigma$ with confidence level 1-$\alpha_1$. In our case we have to test hypothesis:

$H_1$: that $\sigma_{\min} \leq \sigma = \sigma_0 + r \cdot \sigma_0$ where $\sigma_0$ is 100m for R and 1° for $\theta$ with confidence level $\beta=1-\alpha=0.95$.

After calculation we plotted curve of needed number of measurement, i.e. experiment replications to test our hypothesis in figure 5. Also we assess mean error interval where $\mu_1, \mu_2$ are lower and higher value of assessment interval with confidence level $\beta=1-\alpha=0.95$. Also, in our case we have to test hypothesis:

$H_2$: that $\mu \leq \mu_{\max} = \mu_0 + r \cdot \mu_0$, where $\mu_0$ is 100m for R and 1° for $\theta$ with confidence level, $\beta=1-\alpha=0.95$.
Let \( n_1 \) is required sample size for hypothesis \( H_1 \) and \( n_2 \) is required sample size for hypothesis \( H_2 \) then to estimate required sample size to test both hypothesis is \( N = \max(n_1, n_2) \). It is obvious that \( N \) depends on \( m \), \( s \) and \( N \leq 110 \).

Result of this analysis told us that we could expect high number of experiment replication regardless we apply simulation or physical experiment. In order to minimize number of experiments we applied experimental design theory. We identified five input variables (factors) i.e. \( x_1 \) – target speed [m/s], \( x_2 \) – target detection probability [0,1], \( x_3 \) – target radial acceleration [g], \( x_4 \) – extractor error of \( r \) i.e. \( \mu_r \), \( x_5 \) – extractor error of \( \theta \) i.e. \( \mu_\theta \), \( \sigma_\theta \) [mrad], their ranges such is \( x_1 \in [0,300] \), \( x_2 \in [0.7,0.9] \), \( x_3 \in [0.6] \), \( x_4 \in [60,100] \), \( x_5 \in [9,17] \) and variation intervals \([75,0.05,1.5,10,2]\) respectively, five levels of variation (-2,-1,0,+1,+2) and output variables. Our goal was to find out a mathematical model in the form of a second order polynomial as mathematical description of a research subject (factor influence to quality of automatic target tracking) with an adequate precision. In this case we will find out requirement specification for extractor algorithm precision, K- design parameter of maneuver detector in order to minimize target tracking error with minimal number of experiments whether we apply physical or simulation trial. In order to get a mathematical model in the form of a second order polynomial with an adequate precision we applied SECOND ORDER CENTRAL COMPOSITE ROTATABLE DESIGNS (CCRD - Box’s design) which require \( N=32 \) trials, with \( n_0 =6 \) half replica in “NULL” point, instead of \( 5^4=3125 \) trials that full factorial design plan requires. Regression coefficients in case of CCRD \( 2^5-1 + 2 \times 5 + 6 \) are calculated according to next formulas, where, \( k=5 \) number of factors, \( N \) - total number of trials, \( y_i - u_i \) trial response, \( n_j = 2^{5-1}=16 \), \( X_{iu} \) - coded \( i^{th} \) factor values in \( u^{th} \) trial:

\[
\begin{align*}
  b_{ij} &= a_i \sum_{u=1}^{n_j} X_{iu} \cdot y_u \\
  b_{i} &= a_i \sum_{u=1}^{n_j} X_{iu} \cdot y_u
\end{align*}
\]

\[
\begin{align*}
  b_{ii} &= a_j \sum_{i=1}^{N} X_{ii}^2 \cdot y_i \\
  b_i &= a_j \sum_{i=1}^{N} X_{ii} \cdot y_i - a_7 \sum_{u=1}^{N} y_i
\end{align*}
\]

and \([a_1; a_2; a_3; a_4; a_5; a_6]\) = 

\[
[0.1591; 0.0341; 0.0417; 0.0625; 0.0312; 0.0028; 0.0341].
\]

Lack of fit of the obtained regression model, for the case of rotatable designing with trials replicated only in design center, is checked by relations applying Fisher’s criterion i.e. calculate \( F_r = \frac{S_{AD}}{S_{F}^2} \) and find \( F_T \) -tabular value of Fisher’s criterion if degrees of freedom are \( f_{AD}=32-21-5=6; f_E=n_0-1=6-1=5 \) and \( 1-\alpha=95 \% \), where:

\[
S_{AD}^2 = \frac{S_R^2 - S_E}{f_{AD}} = \frac{\sum_{i=1}^{N} (y_i - y_i)^2 - \sum_{j=1}^{n_0} (y_{oj} - \bar{y}_0)^2}{N - (k+2)(k+1)/2 - (n_0 - 1)}
\]

\[
S_{F}^2 = \frac{\sum_{j=1}^{n_0} (y_{oj} - \bar{y}_0)^2}{n_0 - 1}
\]

If \( F_T > F_R \), we may consider the regression equation (1) adequate. Significance of regression coefficients is checked by expressions (2).

\[
\begin{align*}
  \Delta b_0 &= \pm 0.798 \cdot S_T; \Delta b_i &= \pm 0.408 \cdot S_T; \\
  \Delta b_{ij} &= \pm 0.369 \cdot S_T; \Delta b_{i} &= \pm 0.5 \cdot S_T
\end{align*}
\]

The study has identified a hierarchical approach to apply these capabilities in early decision situations, M&S and measurement driven approach to continue to apply the capabilities across the software and system lifecycle. The system test team took the approach of developing a family of simulations and physical experiments to predict and confirm system performance. For every feasible experiment (live or simulation) to get above mentioned information we calculated or estimated required time, cost and importance for various test scenarios to find optimized resources allocation T&E model. We did what-if analysis and find out that optimized STP scenario, in short, was:

1. Model extractor algorithm through simulation, based on assumption that number of reflected radar pulses fitted in \( N^{th} \) range gate has an optional distribution (the first credible option is Normal distribution, then Weibull, Uniform etc.) and satisfy target “echo end” criteria described in [15]. Same assumption is that intensity of reflected radar pulses for azimuth estimation of target center has Normal distribution because of antenna beam shape. Error of range and azimuth estimation should have Normal distribution.
2. Design simulation based experiments of target tracking algorithm for full-scale attack according to
various target trajectories, speed and environment characteristics following CCRD - Box’s experimental design as planned above in this section.

3. Calibrate extractor algorithm during ATTRS integration test phase using hardware target simulator (Doppler transponder). Iterate M&S based on experiment results.

4. Validate and calibrate ATTRS in field test for full-scale real target attack according to various target trajectories, speed and environment characteristics applying OART to select minimal test cases. Iterate M&S based on experiment results.

The highest level simulation predicted the performance of the entire system to a full-scale attack. To facilitate the design of the simulation, the models of subsystems (extractor and Kalman tracking filter) were only as detailed as required to enable the system simulation to model overall system performance. Detailed simulations of all major subsystems validated the high-level models. In some cases, the phenomena modeled in those subsystem simulations were based on even more detailed simulations accounting for the fundamental physics involved. We determined regression coefficients of equation (1) with adequate precision i.e. Fisher’s criterion $F_{T} > F_{R}$ was satisfied. Also, we determined that all included factors and their 2-factor interactions have statistically significant influence on tracking precision and higher number (3 to 5) of factor interactions are not statistically significant. A test factor is a system condition or parameter that has several options that need to be tested. Testing each option of each test factor once will find all single-mode faults which are faults not depending on any other parameters to occur. Some test factors interact incorrectly with one or more other test factor options causing multi-mode faults. The objective in combinatorial testing is to identify a minimum set of tests that will find all serious multi-mode faults. Often, all 2-way (pairs) and 3-way combination of parameters will find most multi-mode software faults. Identifying a minimum set of tests that check each parameter interacting with every other parameter (i.e., all pairs of parameters) is often a very difficult venture if pursued in an ad hoc, non-systematic fashion. Orthogonal arrays (OAs) provide a systematic means for identifying a minimal set of highly effective tests. Unfortunately, some training in combinatorial software testing techniques available in the industry today is not very helpful in teaching this.

6 Conclusions

Quality assurance measures, such as testing during the software development process, are recognized software engineering methods and are included in every software process model. In practice it is important to design in software product quality in each software development step, from beginning to the product delivery and maintenance phase. Software testing is used to validate the correctness of programs as well as to weed out defects, and is typically the most expensive phase of the production process. Thus the problem of performing effective and economical testing is a key issue. Few techniques for reduction of a number of test cases that assure required confidence level in software testing are investigated in this paper. 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. To solve this problem, we revisit the reduction hypotheses, such as: uniformity, regularity, induction, deduction, analogy etc., combined with design of experiments (combinatorial and orthogonal vectors design) for test generation and show that the size of the test suite which these techniques generates is fewer order of magnitude factor smaller in size than any other traditional test suite generation method or generate near optimal test suite size. A relatively low degree of $n$-way combinations of values would detect nearly all errors in the database. Appropriate levels of $n$ could be $3 \leq n \leq 6$, according to dependability requirements, suggesting that combinatorial testing would be effective for this type of software. If experience shows that all errors in a particular class of software are triggered by combinations of $n$ values or less, then testing all combinations of $n$ or fewer values would provide a form of “pseudo-exhaustive” testing. Since most variables are likely to have a very large range of values, equivalence classes would need to be used in practice. Because the effectiveness of combinatorial testing depends on the fact that a single test case can include a large number of pairs (or higher degree combination) of values, this approach would not be effective for most real-time or other software that depends on testing event sequences, but it may be applicable to subsystems within real-time software. Empirical studies of other classes software would be helpful in evaluating the applicability of combinatorial testing.

References


