# Scientific Test & Analysis Techniques for Software Testing

*Authored by:*

*Francisco Ortiz, PhD*

*17 April 2015*

*Revised 11 October 2018*

**The goal of the STAT COE is to assist in developing rigorous, defensible test strategies to more effectively quantify and characterize system performance and provide information that reduces risk. This and other COE products are available at [www.AFIT.edu/STAT](www.AFIT.edu/STAT).**

STAT Center of Excellence
2950 Hobson Way – Wright-Patterson AFB, OH 45433

# Table of Contents



*Revision 1, 11 Oct 2018: Formatting and minor typographical/grammatical edits.*

## Executive Summary

Defense systems in development are increasingly becoming software dependent. It has been estimated that more than 50% of software development time will be used for testing (Kuhn et al., 2010). Developing efficient, yet rigorous test strategies for software intensive systems is paramount to mitigate cost growth and schedule slips. This best practice addresses how scientific test & analysis techniques (STAT) can be applied to software testing. The underlying STAT process (Plan, Design, Execute, and Analyze) remains the same no matter what system is under test. What can differ, however, are the tools used within the process, particularly in the design and analyze phases. The reason these tools differ is due to the nature of the performance measures (e.g., binary and deterministic rather than continuous and random) and how the risk associated with a designed experiment is measured (e.g., based on coverage of operational space rather than the error associated with inference). Two test design approaches that have been used frequently in the software testing field are combinatorial and risk based designs. It is not only imperative to understand how these tools and approaches work, but also to understand the environment/scenario in which their use is appropriate.

**Keywords:** Combinatorial designs, factor covering arrays, design of experiments

## Introduction and Background

### Software Testing Definitions

Software testing is the process of evaluating a system or its components with the intent to determine if the software satisfies specified requirements. ANSI/IEEE 1059 (1993) defines software testing as "a process of analyzing a software item to **detect** differences between existing and required conditions, that is **defects/errors/bugs,** and to **evaluate the features** of the software item." So essentially, software testing can be viewed as an exhaustive functional test of a system by executing all possible combinations of inputs within a system.

In the software testing community, the term "positive" and "negative" testing is often used to describe the goals of testing. Positive testing can be viewed as testing the "shall do" functions of a system. Given the correct inputs, we want to make sure software works as expected. Tests for a physical system usually have this same objective. Negative testing examines the "shall NOT do" functions of a system. The goal in negative testing is to make sure the system does not do something it should not do. For example, suppose we test a student web portal that allows students to see their grades. We would want to make sure the system does not grant the students permission to change their grades. Negative testing answers the question: Given incorrect inputs, does the system behave in an unexpected way? Negative testing is somewhat unique to software testing. When testing physical systems, we typically are not interested in trying to trick or break a system by providing inputs outside standard operating conditions. Including these types of inputs leads to an expansion of test scope and more consideration and more exploration of the input space must be taken to thoroughly vet the system under test.

## STAT Process

Regardless of the system under test, the overall STAT process is the same; what will differ across systems are some of the tools used within the process. The STAT methodology (shown in Figure 1) is an iterative process that begins with the requirement and proceeds through the generation of test objectives, designed experiments, and analysis plans all focused on definitively addressing the requirement. For a detailed description of the STAT process, see *Guide to Developing an Effective STAT Test Strategy* (2017).

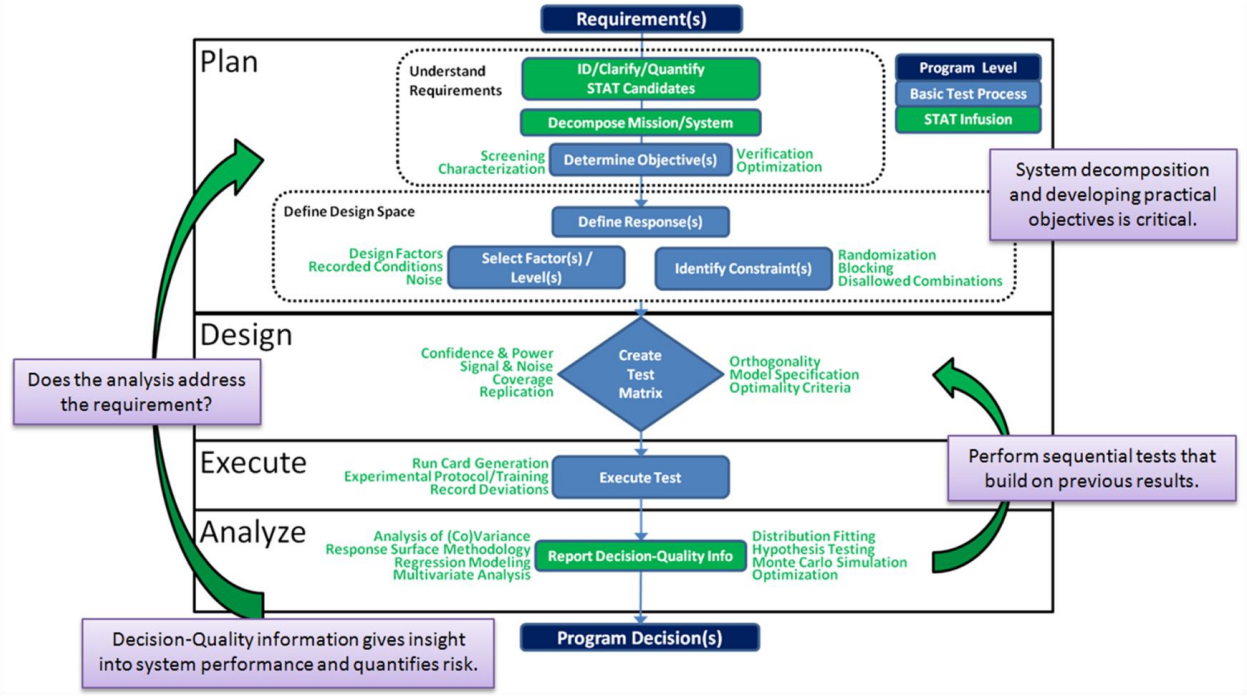


**Figure 1: The STAT Process**

## Performance Measures in Software Testing

Understanding the nature of the performance measures (responses) being collected helps determine the correct tools and approach to add rigor in testing the system. In the following subsections, we briefly introduce some examples of performance measures that can be used to assess a software-intensive system.

### Continuous and Stochastic Measures

If the performance measure in the software system is continuous and there is some variation in the system, even a small amount, then the testing approach and tools used are the same as those used when testing a physical system. Some examples of continuous and stochastic performance measures for software intensive system are message completion rate (%) and message latency (seconds). In these cases, we recommend using design of experiments (DOE) to build the test cases/runs and then perform

linear regression analysis with statistical intervals to build a model and estimate/predict performance across the operational space.

For example, suppose we want to test the requirement that message completion rate never falls below 80% across the operational space. A test design would vary potentially influential factors, say number of concurrent users and file size, to see if there are areas in the operational space where performance is below specification (<80%). The results from a test could be a regression model like the one shown in Figure 2 that shows how performance varies as a function of the input factors in both a response surface plot (Figure 2a) and contour plot (Figure 2b). Note that both number of users and file size are shown in coded units and therefore range from $-1$ to 1.

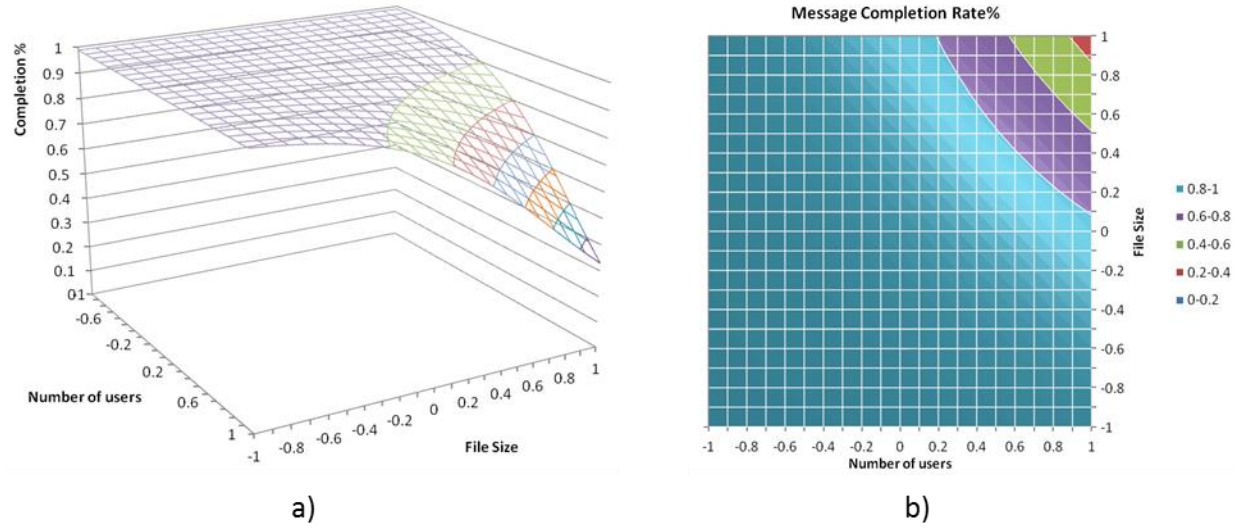


**Figure 2: Regression model for message completion rate as a function of number of users and file size a) response surface and b) contour plot**

Notice because the input space is continuous, a data point that is failing to meet the specification implies that the surrounding area (neighborhood) may fail as well. You can see that that the system begins failing specifications when file size and the number of users are at their high settings. For more information on using DOE and statistical analysis to assess system requirements, see Burke et al. (2017).

## Qualitative Measures

There will be circumstances in which qualitative measures are the only way to assess a software-intensive system. For example, consider a situation in which the goal of a test is to compare a new software user interface (UI) versus the legacy UI to determine if it improves usability. One approach to collect data would be to survey users. The survey must have carefully worded statements and ask the user how much they agree with the statement based on a Likert-scale such as the one shown in Figure 3.

| Strongly Disagree | Disagree | Undecided | Agree | Strongly Agree |
|---|---|---|---|---|
| (1) | (2) | (3) | (4) | (5) |

**Figure 3: Example of a Likert-scale**

There is a science in creating a sound survey. Please see Grier (2013) for a best practice on using survey for test and evaluations.

## Binary and Deterministic Measures

More often, the intent of a software test is to find defects/errors/bugs in a system. The performance measure in this case is actually deterministic and binary (pass/fail). To illustrate, consider the following Microsoft Word example taken from Kuhn (2009). For this test, the goal is to examine all font effects (shown in Figure 4) and ensure that they get displayed properly on the screen.

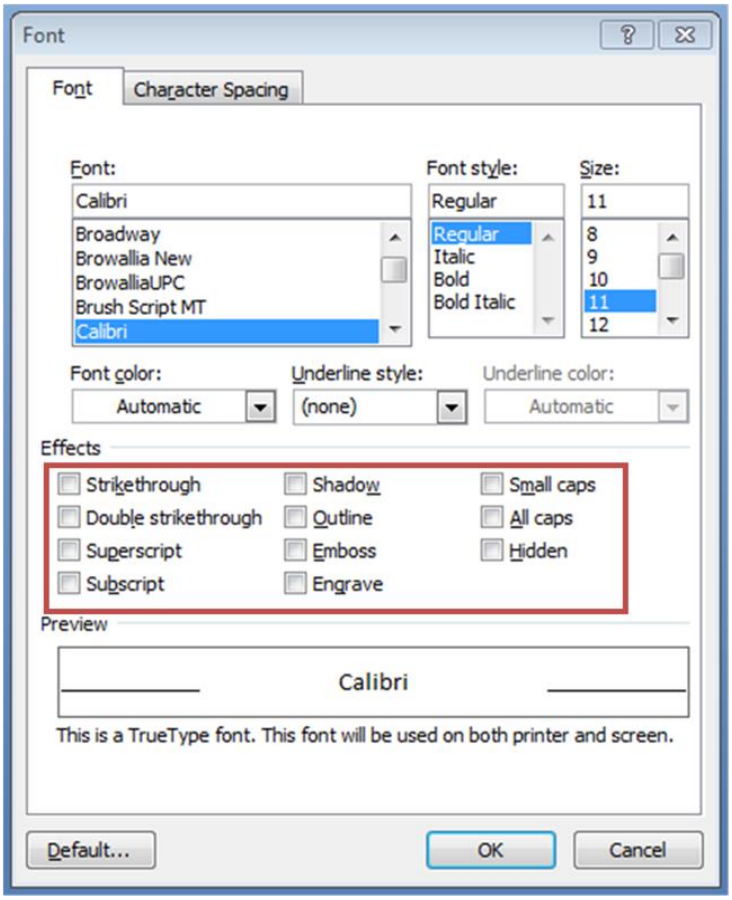


**Figure 4: Microsoft font effects**

Each input is binary (on and off). If we want to test every possible combination, we would need $2^{11}$ (2048) trials. If we were to test only 2-way combinations of the inputs there would be 121 tests to examine (see Figure 5).

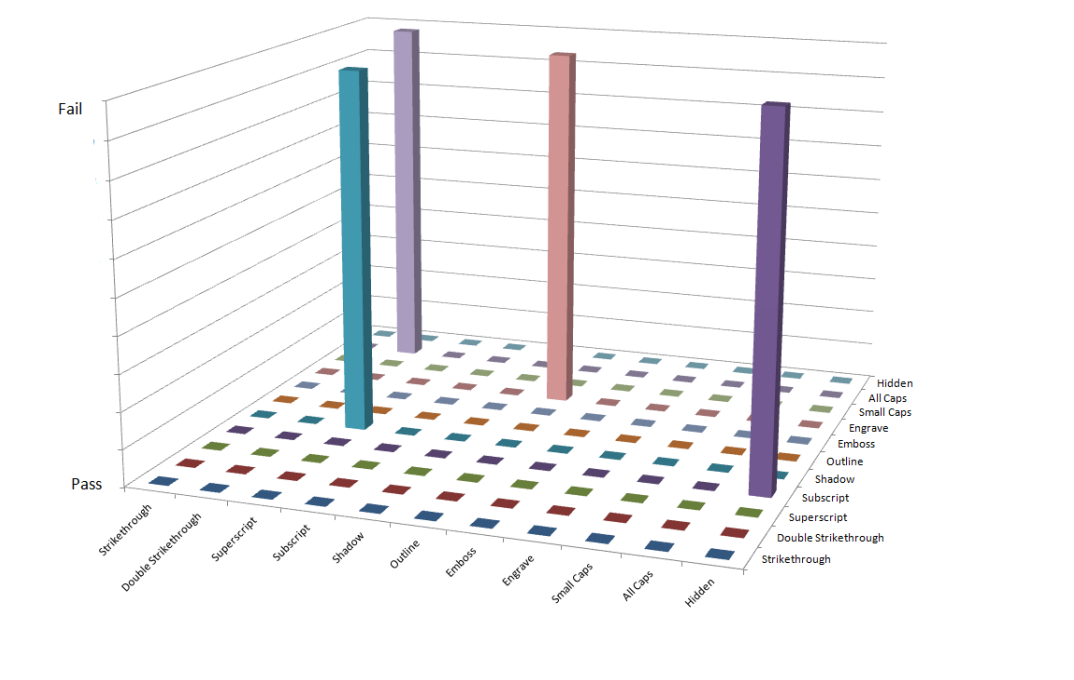


**Figure 5: Operation space for font effects test**

Notice that the data types for input factors are categorical and there is no relationship between factor levels/settings. A failure at one test point does not mean that its nearest neighbors will fail as well compared to the example shown in Figure 3. The input factors here can be viewed as switches and we are trying to find which combinations of switches result in a failure.

Due to the different nature of the data being collected, traditional methods/measures used to evaluate a designed experiment do not directly apply. Statistical power and confidence are two common measures used to evaluate the adequacy of a test design when the response is continuous and stochastic. Power and confidence are calculations of the probability of making an incorrect inference about the system under test (Type II and Type I error, respectively). These measures assume a stochastic system which obviously is not the case with some software tests. When the performance measure is deterministic, test design evaluation is based more on the concept of coverage. For cases like this, we recommend employing a factor covering array design (e.g., combinatorial t-way or risk based design) to efficiently and quickly identify errors in the system. We discuss these types of designs in the following sections.

# Tools and Approaches

## Combinatorial Designs

In software testing, any particular combination of input factor settings could trigger a fault. Therefore, every possible combination of input factors would have to be investigated in order to ensure all faults or bugs have been detected. Consider the following flight reservation website example. Table 1 lists the input factors and the number of levels/settings that need to be tested to perform a full evaluation of the operational space. There are 7,338,354,278,400 possible combinations of input factors. This is obviously time consuming, costly, and impractical to execute.

**Table 1: Input space for a flight reservation website example.**

| Factors | Levels |
|---|---|
| Browser | 5 |
| As Much as Possible, Fill in Information | 3 |
| As Much as Possible, Navigate Using | 2 |
| Impersonate a User that is | 3 |
| Adults | 2 |
| Seniors | 2 |
| Children | 2 |
| Trip Information | 3 |
| Include Hotel | 2 |
| Include Car | 2 |
| Flexible Dates | 2 |
| Leaving From (City or Town) | 4 |
| Leaving From (Area) | 6 |
| Going to (City or Town) | 4 |
| Going to (Area) | 6 |
| Class of Travel | 3 |
| Purchasing How Far in Advance | 4 |
| Saturday Stay-over | 2 |
| Sort by | 5 |
| Airline Preference | 4 |
| Change Search - Nonstop flights only | 2 |
| Change Search - Refundable Flights Only | 2 |
| Change Travelers | 3 |
| Preview Seat Availability | 2 |
| Change Search - Departure Airport | 2 |
| Change Search - Destination Airport | 2 |
| Change Search - Departure Date | 4 |
| Change Search - Return Date | 4 |
| **Total Possible Combinations** | **7.34E+12** |

Luckily, a NIST study (Kuhn et al., 2009) based on historic data collected from various systems of varying complexity found that most faults are caused by the interactions between only a few factors. Figure 6 shows the cumulative percentage of faults found versus the level of interactions between factors for various systems in the NIST study. Over 80% of faults were found by looking at just 3-way interactions, over 90% with 4-way interactions and virtually all with 6-way interactions. The results of this study imply

that there are opportunities for efficient testing in software testing as not all factor interactions need to be investigated.

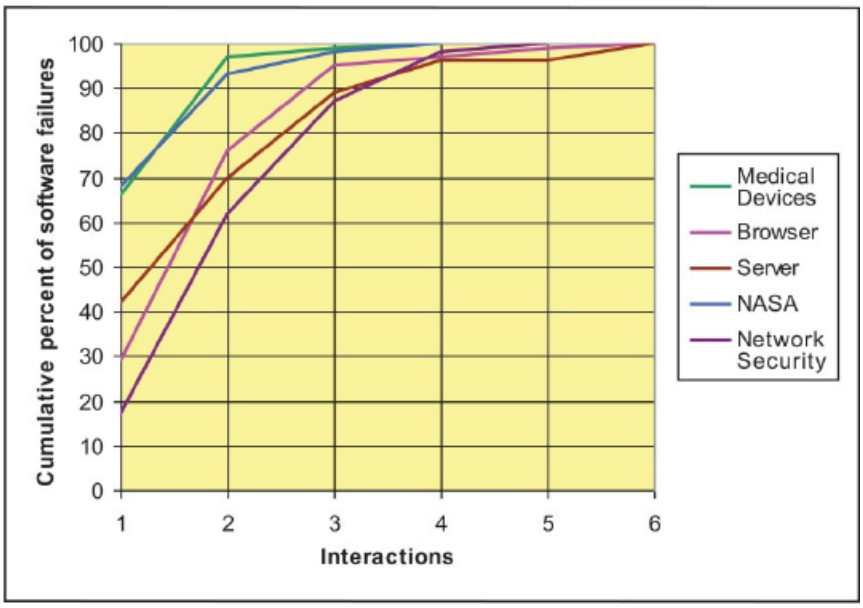


**Figure 6: Historical data from various systems showing the cumulative percent of software failures due to t-way interactions of input factors (Kuhn et al., 2009)**

Combinatorial designs attempt to maximize test coverage by combining factor levels to cover all the t-way combinations in the minimum number of test runs. To illustrate what these designs look like, consider the Microsoft Word example introduced earlier. Let's start off by looking at just two input factors "Shadow" and "Outline" as shown in Figure 7. A value of 1 means that the check box for that font effect was selected while a value of 0 means it was not selected. If we want to look at all 2-way interactions for these two factors, then we need to investigate all $2^2$ cases. There is no other way to be more efficient for a test with just two factors.

**2 variables - All Combinations**

| Case | Shadow | Outline |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1 | 0 |
| 3 | 0 | 1 |
| 4 | 0 | 0 |

**Figure 7: Two factor combinatorial design**

Let's expand this problem by adding "Emboss" as a new factor (see Figure 8). To look at all combinations between the three factors (3-way interactions), we would need $2^3 = 8$ runs.

**3 variables – All combinations**

| Case | Shadow | Outline | Emboss |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 |
| 5 | 0 | 1 | 1 |
| 6 | 0 | 0 | 1 |
| 7 | 0 | 1 | 0 |
| 8 | 0 | 0 | 0 |

**Figure 8: Potential three factor combinatorial design**

However, if we are only interested in 2-way interactions, then the design can be reduced to just 4 runs. Figure 9 shows how the 2-way interactions between the three input factors are all covered in 4 runs.

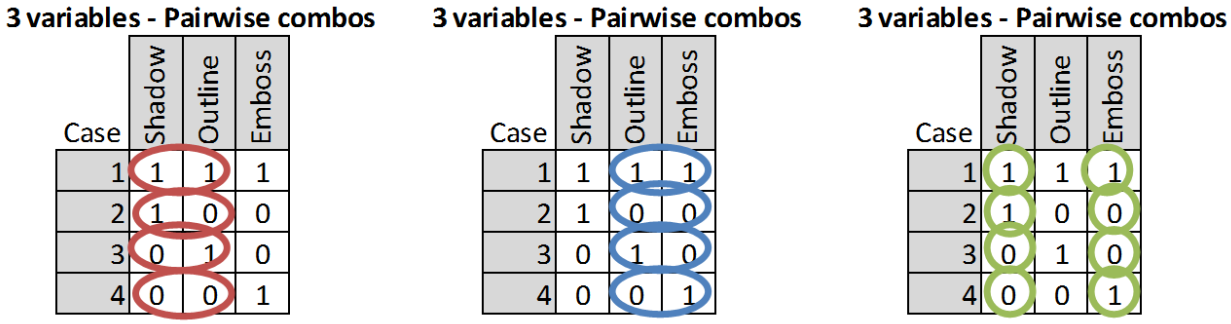


**Figure 9: Three factor design covering all two way combinations.**

For the same amount of runs in the two-factor example, we can examine 3 factors and their 2-way interactions.

Now consider all 11 factors as shown in Figure 10, remembering that to test all combinations would require 1024 runs. All 2-way combinations can be covered in just 10 runs which, based on Kuhn's NIST study, could translate in finding >60% of all faults in the system.

| Case | Strikethrough | Double_Strikethrough | Superscript | Subscript | Shadow | Outline | Emboss | Engrave | Small_Caps | All_Caps | Hidden |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 5 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 10 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

**Figure 10: Eleven factor design covering all two-way combinations in just 10 runs**

For the flight reservation problem mentioned earlier, exhaustive testing would require 7,338,354,278,400 runs. By comparison, all 2-way interactions could be tested in just 41 runs! All 4-way interactions which, based on historic data would find over 90% of faults, can be done in just 1,421 runs. This is a huge amount of savings in terms of time and resources needed to vet a system.

The following is a list of some software available to easily create these designs.

- ACTS: Available at http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html, Java based, free of charge
- Hexawise: Available at www.hexawise.com , web-based, free trial available
- JMP 14: Available at www.jmp.com, free trial available

## Risk Based Designs

A risk based design applies decision analysis in determining which test cases should be run and in what order. This approach has been advocated by the Office of the Secretary of Defense (OSD) in a 2010 memo, *Guideline for Operational Test and Evaluation of Information and Business Systems*. The approach prioritizes what features of a software system and their associated scenarios/cases should be run based on a weighted average of various risk attribute scores. These scores place emphasis on higher level capabilities or functionality required by the software system versus testing each individual requirement. The goal here is to fit test time to given resource constraints.

Suppose we are testing a company's pay and personnel system. Correct salary payments may be more important than correct accrued vacation time or the correct employee start date. Of course, each one of these features is important, but given a very constrained time window to perform testing, the customer would want the salary payment requirements tested thoroughly first.

Let's use a simplified example. Suppose there are 50 requirements that could be tested in this pay and personnel system. For each requirement, stakeholders and subject matters experts will provide a score based on:

- Business Criticality – measures how critical the capability is to the business process
- Failure Probability – indicates how likely a test is to fail based on the capability
- Functional Complexity – indicates the technical difficulty in executing the function (i.e., process or computational steps)

The weighted average of these 3 scores is the overall "risk" score for that particular requirement. We then order the requirements based on that score (see Figure 11). If for example, testing was truncated due to time and we only examined the top 26 requirements we can quantify the risk covered (e.g., 80% in the example shown in Figure 11).

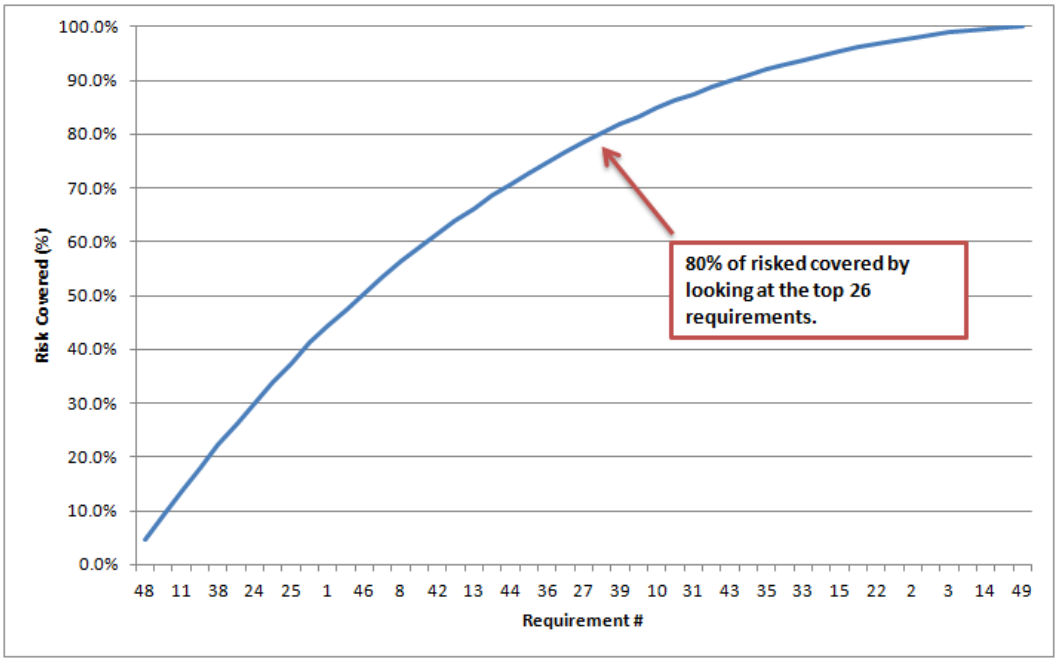


**Figure 11: Cumulative risked covered using a risk based design**

It is important to note that this approach requires stakeholders to agree upon a list of important attributes. Business criticality, failure probability, and functional complexity are among standard measures used in industry, but this is not an exhaustive list by any means.

## Using Combinatorial and Risk Based Designs Together

Using risk based and combinatorial designs together can be a very powerful tool in introducing defensible rigor to software testing. A risk based design could be used to rank requirements and then evaluating each requirement could entail generating one or more combinatorial test designs. For example, let's say there are 50 requirements and each requirement is evaluated on a scale from 1 to 10 based on business criticality, failure probability, and functional complexity where a high score indicates high risk. Table 2 shows an example of this scoring for the 50 requirements. The requirements have been sorted from highest risk to lowest risk.

**Table 2: Example using both combinatorial and risk based designs together (27 of 50 requirements shown)**

| Req. Test Order | Req. | Business Crit. | Failure Prob. | Functional Comp. | Risk Score | Risk Category | Req. T-way coverage | Size (runs) | Runs Executed | % Covered | Individual Risk % | Cumulative Risk % |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 14 | 9 | 10 | 10 | 96.5 | High | 4-way | 145 | 145 | 100.0% | 5.8% | 5.8% |
| 2 | 7 | 9 | 9 | 9 | 90.0 | High | 4-way | 325 | 325 | 100.0% | 5.4% | 11.3% |
| 3 | 24 | 9 | 7 | 9 | 90.0 | High | 4-way | 428 | 428 | 100.0% | 5.4% | 16.7% |
| 4 | 5 | 9 | 9 | 9 | 82.8 | High | 4-way | 111 | 111 | 100.0% | 5.0% | 21.7% |
| 5 | 29 | 7 | 8 | 9 | 88.8 | High | 4-way | 744 | 744 | 100.0% | 5.4% | 27.1% |
| 6 | 10 | 10 | 7 | 10 | 71.6 | High | 4-way | 738 | 738 | 100.0% | 4.3% | 31.4% |
| 7 | 35 | 6 | 9 | 7 | 71.4 | High | 4-way | 251 | 251 | 100.0% | 4.3% | 35.7% |
| 8 | 19 | 5 | 10 | 7 | 65.1 | High | 4-way | 345 | 345 | 100.0% | 3.9% | 39.7% |
| 9 | 12 | 7 | 10 | 3 | 63.4 | High | 4-way | 807 | 807 | 100.0% | 3.8% | 43.5% |
| 10 | 4 | 10 | 10 | 5 | 53.5 | High | 4-way | 559 | 559 | 100.0% | 3.2% | 46.7% |
| 11 | 41 | 10 | 8 | 3 | 67.5 | Medium | 3-way | 23 | 23 | 100.0% | 4.1% | 50.8% |
| 12 | 37 | 6 | 5 | 5 | 56.9 | Medium | 3-way | 28 | 28 | 100.0% | 3.4% | 54.2% |
| 13 | 17 | 6 | 3 | 10 | 49.7 | Medium | 3-way | 25 | 25 | 100.0% | 3.0% | 57.2% |
| 14 | 40 | 10 | 3 | 6 | 54.9 | Medium | 3-way | 35 | 35 | 100.0% | 3.3% | 60.6% |
| 15 | 31 | 8 | 3 | 4 | 51.7 | Medium | 3-way | 35 | 35 | 100.0% | 3.1% | 63.7% |
| 16 | 13 | 5 | 9 | 6 | 39.5 | Medium | 3-way | 40 | 40 | 100.0% | 2.4% | 66.1% |
| 17 | 15 | 6 | 6 | 4 | 39.5 | Medium | 3-way | 31 | 31 | 100.0% | 2.4% | 68.5% |
| 18 | 34 | 9 | 5 | 5 | 42.6 | Medium | 3-way | 40 | 40 | 100.0% | 2.6% | 71.0% |
| 19 | 9 | 9 | 8 | 5 | 37.2 | Medium | 3-way | 34 | 34 | 100.0% | 2.2% | 73.3% |
| 20 | 23 | 4 | 9 | 9 | 36.7 | Medium | 3-way | 31 | 31 | 100.0% | 2.2% | 75.5% |
| 21 | 2 | 10 | 6 | 10 | 27.5 | Medium | 3-way | 34 | 34 | 100.0% | 1.7% | 77.2% |
| 22 | 22 | 6 | 10 | 1 | 28.6 | Medium | 3-way | 38 | 38 | 100.0% | 1.7% | 78.9% |
| 23 | 11 | 4 | 2 | 10 | 25.9 | Medium | 3-way | 36 | 36 | 100.0% | 1.6% | 80.4% |
| 24 | 39 | 9 | 3 | 4 | 28.9 | Medium | 3-way | 24 | 24 | 100.0% | 1.7% | 82.2% |
| 25 | 1 | 4 | 3 | 5 | 23.5 | Medium | 3-way | 38 | 38 | 100.0% | 1.4% | 83.6% |
| 26 | 3 | 1 | 9 | 10 | 19.6 | Medium | 3-way | 26 | 20 | 76.9% | 0.9% | 84.5% |
| 27 | 21 | 7 | 1 | 5 | 22.4 | Medium | 3-way | 27 | 0 | 0.0% | 0.0% | 84.5% |

What level of rigor or strength should be associated with the underlying combinatorial design? One solution is to use the requirement risk scores to help answer that question. Requirements with high risk scores (Column "Risk Category") would suggest using a combinatorial design that would cover higher t-way interactions (e.g., all 4-way or 5-way interactions between input factors), medium risk scores use just 3-way interactions, and low risk scores could probably be satisfied with just 2-way interactions. Now say, schedule constraints leads to testing being stopped after requirement 26, and only 20 of the 26 runs of the combinatorial test design were executed (see line 26 in Table 2). We only covered 76.9% of all 3-way interactions for that individual requirement. The cumulative risk covered for the entire test strategy

is calculated to be 84.5%. The end result of using this approach is not only a test strategy that optimizes resources and time within constraints, but also a quantifiable measure of risk coverage.

## Conclusion

STAT can be applied to software testing. Regardless of the system under test, this STAT process should be followed using these basic steps:

1. Decompose requirements.
2. Identify test objectives.
3. Create test designs that quantify risk.
4. Perform analysis that results in decision quality information.

Software testing may require some unique tools to be utilized due to the nature of the performance measure of interest and the expanded scope of the test objectives. Combinatorial designs and risk based designs may be better suited for some software tests since they are based more on obtaining coverage of the operational space rather than the error associated with statistical inference. Combinatorial designs aim to maximize test coverage by creating an optimal (minimum run) design that covers all t-way combinations. Risk based designs use a decision analysis approach to determine and prioritize which test cases should be executed first. Both approaches can be used in conjunction to better communicate the rigor associated with the overall test strategy of a program.

## References

Burke, Sarah et al. “Guide to Developing an Effective STAT Test Strategy V5.0,” Scientific Test and Analysis Techniques Center of Excellence (STAT COE), Dec 2017.

Gilmore, J. Michael. "A statistically rigorous approach to test and evaluation (T&E)." ITEA Journal, vol. 34 2013, pp. 225-229.

Grier, Rebecca A., “Surveys in Test and Evaluation”, IDA Document NS D-4934. 2013.

Institute of Electrical and Electronics Engineers. *Guide for Software Verification and Validation Plans*. IEEE Std 1059-1993.

Kuhn, Rick, et al. “Combinatorial Software Testing.” Computer (IEEE Computer Society), vol. 42, no. 7, 2009, pp. 94–96., doi:10.1109/mc.2009.253.

Kuhn, D.R, Kacker, Raghu N., Lei, Yu, “Advanced Combinatorial Test Methods for System Reliability”, *Reliability Society Annual Technical Report*, 2010.

Kuhn, D.R, Wallace, D.R, Gallo, Jr., AJ, “Software Fault Interactions and Implications for Software Testing,” IEEE Trans. on Software Engineering, vol. 30, no. 6, 2004.

National Institute of Standards and Technology, Computer Security Division, http://csrc.nist.gov/groups/SNS/acts/index.html.

Office of the Secretary of Defense, “Guideline for Operational Test and Evaluation of Information and Business Systems”, Sept. 2010