STAT COE-Report-39-2018

Software Reliability Fundamentals for Information Technology Systems

Authored by: Bill Rowell, PhD and Kyle Kolsti, PhD

9 November 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.

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

Table of Contents

Executive Summary

To be effective, software reliability programs for federal government Information Technology (IT) systems must focus on a number of key goals:

- Establishing common, meaningful software reliability definitions among all program participants
- Developing, implementing, and tracking useful software reliability metrics
- Implementing processes and supporting organizational structures in the development environment to efficiently and effectively
    - Identify, remove, and fix defects
    - Capture, process, and display failure and defect data
- Understanding and capturing key reliability aspects of the operational environment

This paper lays out fundamental practices that the leadership of federal government IT software reliability programs can adopt to achieve these goals.

Keywords: Software, reliability, information technology

Introduction

Department of Defense (DoD) IT systems are comprised of National Security Systems (NSS) and Defense Business Systems (DBS). Typically, NSS encompass telecommunications or information systems involving intelligence activities, cryptologic activities related to national security, command and control of military forces, or equipment that is an integral part of a weapon system. On the other hand, DBS typically involve systems performing financial, contracting, planning and budgeting, installations management, human resources management, or training/readiness. DBS are software intensive and do not contain separate hardware components that perform any function other than hosting the software that performs the business function. The breakdown of DoD IT systems into hardware/software vs. software-intensive categories is also reflected in IT systems procured by the Department of Homeland Security (DHS) and other federal agencies. Regardless of the category of IT system, implementing an efficient, effective software reliability program is critical to the accomplishment of the mission of federal government IT systems.

Developing a reliability program for the hardware components of an IT system is a well-established, widely understood discipline. On the other hand, developing one for the software components is more challenging because key concepts of software reliability are often not known, not well understood, or confused with those of hardware components. Not surprisingly, IT acquisition programs vary greatly in their approaches to software reliability programs. Some adopt hardware-based reliability approaches and consequently struggle to find useful metrics for measuring progress whereas others adopt efficient, effective software reliability metrics and methods that drive successful decision making.

Although we are aware of a variety of philosophies and techniques for implementing a software reliability program, we will take a pragmatic approach focusing instead on the fundamental best practices that we believe need to be a part of every software reliability program. This paper systematically lays out the best practices essential for an IT system software manager to incorporate into a federal government IT software reliability program. After discussing the key definitions that should be shared by all members of the software reliability team and formulating and tracking software reliability metrics, we discuss reliability best practices critical to a federal IT system's development and operational environments.

Key Definitions

The first recommendation for a software reliability program is to adopt authoritative definitions from established sources (for example, IEEE standards, which are available by subscription from IEEE.org and cited in this document). These and similar authoritative definitions provide a common basis for communicating effectively among all parties to the acquisition system development. We encourage programs to reference these definitions in early program documentation (like requirements and Requests for Proposals) and use these kinds of definitions as the foundation for developing future documents as well as in their reliability discussions. Early adoption of a coherent set of authoritative definitions will minimize confusion/misunderstandings throughout the software reliability program.

Table 1 displays the IEEE 1633 definitions for software reliability, software reliability engineering as well as software quality. A few points are in order. Per its definition, software reliability engineering applies to both the development and operational environment. Software quality and reliability have significantly different definitions: software quality is associated with how the functionality of the software compares with some known standard (specifications, attributes, expectations, needs) while reliability focuses on the period of time the software continues to operate under specified conditions in an acceptable manner (usually without failure). Understanding the difference between these two definitions will come in handy when we discuss how the focus of software reliability needs to change between the development and operational environments.

Table 1: Fundamental Software Quality/Reliability Definitions

Term	IEEE 1633 Definition
Software Reliability (SR)	**(A)** The probability that software will not cause the failure of a system for a specified time under specified conditions. **(B)** The ability of a program to perform a required function under stated conditions for a stated period of time. NOTE—For definition (A), the probability is a function of the inputs to and use of the system, as well as a function of the existence of defects in the software. The inputs to the system determine whether existing defects, if any, are encountered.

Software Reliability Engineering (SRE)	**(A)** The application of statistical techniques to data collected during system development and operation to specify, estimate, or assess the reliability of software-based systems. **(B)** The application of software reliability best practices to enhance software reliability characteristics of software being developed and integrated into a system.
Software Quality (SQ)	(**A**) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs, such as conforming to specifications. (**B**) The degree to which software possesses a desired combination of attributes. (**C**) The degree to which a customer or user perceives that software meets the user's composite expectations. (**D**) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer. (**E**) Capability of the software product to satisfy stated and implied needs when used under specified conditions.

Each software reliability program must precisely define the conditions under which the software will operate as well as what constitutes an "unreliable" event (software failure). These definitions are necessary for ensuring process consistency and fostering a common understanding among the vendors, contracting officers, and acquisition program personnel. Consequently, products and deliverables should be using these terms in a manner consistent with the formal program definitions. For example, guidance in both DoD and DHS calls for Test and Evaluation Master Plans (TEMPs) to include failure definitions and scoring criteria.

Table 2 provides IEEE 1633 definitions for commonly used (but not always commonly understood) software terms ("error", "defect", "fault", "failure", and "problem"). Other words like "bug" and "issue" may be encountered, but they should be studiously avoided when talking about software reliability or clearly defined in program documentation for their intended usage. In common English all of these terms are more or less synonymous; however, the preciseness of definitions such as those in Table 2 serves to distinguish among these frequently-used terms in a way that enables clear, effective, and efficient conversations among individuals involved in a software reliability program based on them.

Table 2: Definitions Related to Defects and Failures

Term	**IEEE 1633 Definition**
Error	A human action that produced an incorrect result, such as software containing a fault.
Defect	A problem that, if not corrected, could cause an application to either fail or to produce incorrect results. NOTE—For the purposes of this standard, defects are the result of errors that are manifest in the system requirements, software requirements, interfaces, architecture, detailed design, or code. A defect may result in one or more failures. It is also possible that a defect may never result in a fault if the operational profile is such that the code containing the defect is never executed.
Fault	**(A)** A defect in the code that can be the cause of one or more failures.

	(B) A manifestation of an error in the software.
Failure	**(A)** The inability of a system or system component to perform a required function within specified limits. **(B)** The termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. **(C)** A departure of program operation from program requirements. A *failure* may be produced when a fault is encountered and a loss of the expected service results. NOTE 1— A *failure* may be produced when a fault is encountered and a loss of the expected service to the user results. NOTE 2— There may not be a one-to-one relationship between faults and failures. This can happen if the system has been designed to be fault tolerant. It can also happen if a fault does not result in a failure either because it is not severe enough to result in a failure or does not manifest into a failure due to the system not achieving that operational or environmental state that would trigger it.
Problem	**(A)** Difficulty or uncertainty experienced by one or more persons, resulting from an unsatisfactory encounter with a system in use. **(B)** A negative situation to overcome.

Let's look at how these terms are used in a simple example. Assume that a software developer makes an error in the source code, such as coding a numerical division operation where the value of the denominator may have a value of zero but the code doesn't check that the denominator's value has been set to zero and prevent the fault from occurring. Thus, a fault (division by zero) occurs whenever the line of code is executed and the value of the denominator is zero. Depending on how the code handles the fault, the fault may or may not cause a failure, a situation where the expected behavior does not occur. If a fault can cause a failure to occur, then the error in the source code is also a defect. However, if the defective line of code is never executed because the path of the users' requests never includes the line of code containing the defect, failures will not occur even though a defect exists. The more often the operational profile executes the defective line of code, the more frequently failures will occur. A problem arises when the defective line of code prevents the user from receiving the service requested.

Tables 3 and 4 below display the definitions of various attributes of a failure and defect from IEEE Standard 1044, respectively, which could serve as a solid foundation for the program's software reliability data acquisition effort.

Table 3: Definitions of Attributes of a Failure

Attribute	IEEE 1044 Definition
Failure ID	Unique identifier for the failure.
Status	Current state within failure report life cycle. See Table B.1.
Title	Brief description of the failure for summary reporting purposes.
Description	Full description of the anomalous behavior and the conditions under which it occurred, including the sequence of events and/or user actions that preceded the failure.
Environment	Identification of the operating environment in which the failure was observed.

Configuration	Configuration details including relevant product and version identifiers.
Severity	As determined by (from the perspective of) the organization responsible for software engineering. See Table B.1.
Analysis	Final results of causal analysis on conclusion of failure investigation.
Disposition	Final disposition of the failure report. See Table B.1.
Observed by	Person who observed the failure (and from whom additional detail can be obtained).
Opened by	Person who opened (submitted) the failure report.
Assigned to	Person or organization assigned to investigate the cause of the failure.
Closed by	Person who closed the failure report.
Date observed	Date/time the failure was observed.
Date opened	Date/time the failure report is opened (submitted).
Date closed	Date/time the failure report is closed and the final disposition is assigned.
Test reference	Identification of the specific test being conducted (if any) when the failure occurred.
Incident reference	Identification of the associated incident if the failure report was precipitated by a service desk or help desk call/contact.
Defect reference	Identification of the defect asserted to be the cause of the failure.
Failure reference	Identification of a related failure report.

Table 4: Definitions of Attributes of a Defect

Attribute	IEEE 1044 Definitions
Defect ID	Unique identifier for the defect.
Description	Description of what is missing, wrong, or unnecessary.
Status	Current state within defect report life cycle.
Asset	The software asset (product, component, module, etc.) containing the defect.
Artifact	The specific software work product containing the defect.
Version detected	Identification of the software version in which the defect was detected.
Version corrected	Identification of the software version in which the defect was corrected.
Priority	Ranking for processing assigned by the organization responsible for the evaluation, resolution, and closure of the defect relative to other reported defects.
Severity	The highest failure impact that the defect could (or did) cause, as determined by (from the perspective of) the organization responsible for software engineering.
Probability	Probability of recurring failure caused by this defect.
Effect	The class of requirement that is impacted by a failure caused by a defect.
Type	A categorization based on the class of code within which the defect is found or the work product within which the defect is found.
Mode	A categorization based on whether the defect is due to incorrect implementation or representation, the addition of something that is not needed, or an omission.
Insertion activity	The activity during which the defect was injected/inserted (i.e., during which the artifact containing the defect originated).
Detection activity	The activity during which the defect was detected (i.e., inspection or testing).
Failure reference(s)	Identifier of the failure(s) caused by the defect.
Change reference	Identifier of the corrective change request initiated to correct the defect.
Disposition	Final disposition of defect report upon closure.

Metrics

When the key definitions for the software reliability program have been identified, the software reliability program needs to focus its attention on formulating and tracking software reliability metrics. Before addressing these aspects of metrics, it is worthwhile identifying the various decisions that these metrics are expected to support to ensure we are choosing the right metrics and displaying them appropriately. Below is a short list of decisions that properly formulated and computed software reliability metrics can support:

- Is the system ready to proceed to the next stage of its life cycle?
- Which failures should be investigated at this time?
- Which defects should be fixed at this time?
- What is the impact of this failure/defect on the mission of the system?
- What software modules are most in need of attention?
- Does the end-to-end software development process (requirements, design, code, documentation, code fixes) need to be improved?
- What part(s) of the end-to-end software development process need(s) the most improvement?

Formulation

The first sub-section focuses on failure-based metrics while the second on defect-based metrics.

Failures

When estimating hardware reliability failures occurring during the operation of a hardware component, a standard assumption is that the observed failure rate is the output of a stable Poisson process, that is, a statistical distribution with a mean inter-arrival time that doesn't change over time. See Table 5 for IEEE 1633 Standard definition of failure rate. However, the rate of arrival of software failures doesn't depend exclusively on users simply operating the system, such as operating an engine on a motor vehicle, but depends on the rate that users exercise software paths that contain defects and therefore result in software failures. For example, if there are 50 paths that result in faults, the system failure rate is determined by the expected number of user actions that access these paths in a given period of time. Thus, metrics for software reliability must take into account that software failure rates are dependent on the software paths exercised by user requests.

Table 5: IEEE 1633 Failure Rate Definition

Term	IEEE 1633 Definition
Failure Rate	**(A)** The ratio of the number of failures of a given category or severity to a given period of time; for example, failures per second of execution time, failures per month. *Synonym:* failure intensity. **(B)** The ratio of the number of failures to a given unit of measure, such as failures per unit of time, failures per number of transactions, failures per number of computer runs.

The predominant hardware reliability failure-based metric is the Mean Time Between Failures ($MTBF$), which is also called the Mean Time Between Operational Failures ($MTBOF$). $MTBF$ is typically expressed in units that indicate how much use the system has undergone since the last failure, such as hours. Other units of time may be more appropriate; for example, aircraft landing gear may fail based on the number of retract/extend cycles, and the tires may fail based on the number of touchdowns, rather than flight hours. $MTBF$ is the average time it takes from the time the system returns to operation until a failure is experienced, averaged out over the future lifecycle of the system. For software systems a single MBTF number as a measure of reliability isn't as useful as it is for hardware systems and may be misleading. As previously mentioned, for software, a failure only occurs when a user accesses a particular path, whereas for mechanical components, failures are caused by prolonged or repeated operation (wear and tear, fatigue, and so forth). A reported MTBF for a software system must be accompanied by information about the logic paths that were accessed and the severity of the failures that were experienced.

For software reliability the activity we are interested in may be the number of software failures that occur (1) during a nightly regression test or other test runs, (2) while executing a specified number of transactions that may cause the failures of interest, or (3) during some time duration (hour, day, week, month, year, or perhaps user-hours, the sum of times that multiple users are logged in). See Table 6 for IEEE 1633 Standard definitions of time measures. In selecting an activity it's important to know how frequently the activity exercises the software paths associated with the failures of interest.

Table 6: Software "Time" Measures

Term	**IEEE 1633 Definition**
Clock time	Elapsed wall-clock time from the start of program execution to the end of program execution.
Execution time	**(A)** The amount of actual or central processor time used in executing a program. **(B)** The period of time during which a program is executing.
Calendar time	Chronological time, including time during which a computer may not be running.

Defects

Table 7 summarizes three practical software defect-based metrics (Godbole, 2004). Defect Density (DD) measures the defects relative to software size and provides the development organization with data that can be used to help them improve their testing processes. Defect Rate (DR) is the expected number of defects reported over a certain period of time and is important for cost and resource estimates of the maintenance phase of the software life cycle. Finally, there is Defect Removal Efficiency (DRE), which is the efficiency of eliminating defects before delivering the product to the customer and reflects the goodness of the software development process in delivering high quality (defect-free) software to the customer. Of special interest to software reliability programs seeking a metric to incentivize the developer's delivery of high quality software is the assertion of Capers Jones, a well-known specialist in software engineering methodologies and widely published author, that DRE is "the most important single metric for software quality" (Jones, 2011).

Table 7: Software "Defect" Metrics

Term	Definition
Defect Density	$$\frac{\text{[Total number of defects identified during specified testing phase]}}{\text{[Actual Size of the product]}}$$ where size is measured by a suitable measure—most typically either as lines of code (LOC) or Function Points/Features Points (FP)
Defect Rate	$$\frac{\text{[Total number of defects reported]}}{\text{[Length of reporting period]}}$$
Defect Removal Efficiency	$$\frac{\text{[Number of defects identified before delivering product to customer]}}{\text{[(Number of defects identified before delivering product to customer)+(Number of defects identified by the customer within a specified period of time after delivery])}}$$

Tracking

Reliability tracking is the process of collecting data and reporting appropriate metrics to discover trends in software program reliability. It provides insight into the health of the program and gives program managers and decision makers important risk information supporting critical decisions. Figure 1 depicts typical patterns of software reliability metrics (Failures/CPU hour (right vertical axis) and Defects at completion (left vertical axis)) over the course of multiple, consecutive agile sprints. The upward segments in the figure typically reflect activity coding new functionality while the downward segments reflect activity fixing defects.

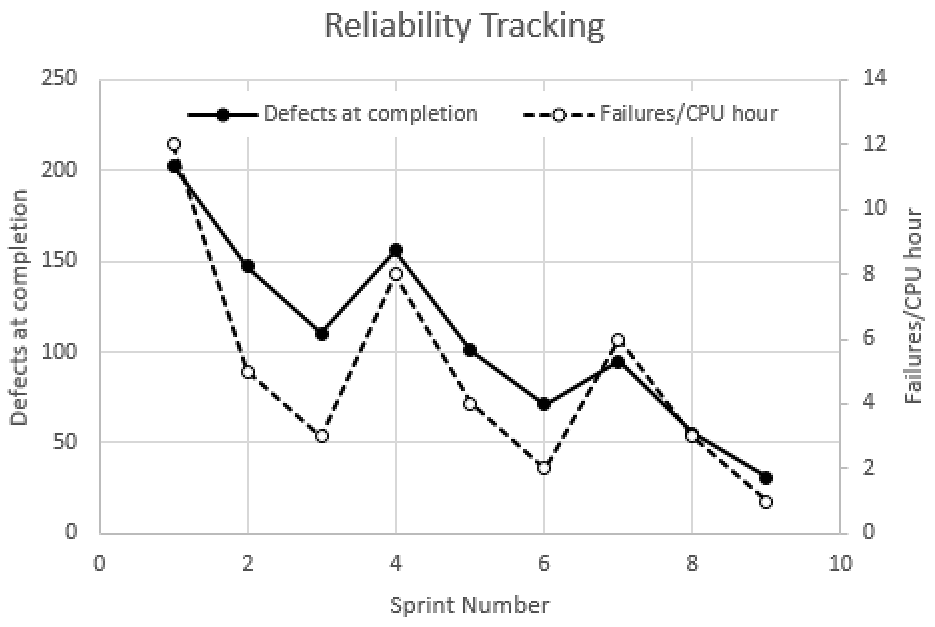

Figure 1: Tracking Software Reliability in an Agile Development Environment

Instituting this process early in the program greatly enhances the program's ability to assess software quality and reliability during development. It is recommended that programs consider writing the contract to include deliverables with metrics that truly enable effective reliability tracking.

During the software development process developers are focused on adding features and capabilities and repairing defects that arise in the new code. We believe it's still worthwhile to track the number of failures and defects at this time, such as those that occur in regression testing following a nightly build, to ensure that development process is under control. However, the focus at this stage should not yet be on meeting reliability goals but rather on evaluating software quality (availability of new capability). Thus, reliability tracking should be more fine-grained in the development environment, for example, tracking failure rates at the module level, to better support identifying the areas of the software that need the most quality improvement.

In the operational environment during the maintenance life cycle phase we are more interested in improving the reliability of a stable code base. Here we would focus on software reliability growth. See Table 8 for the IEEE 1633 Standard definition.

Table 8: Definition of Software Reliability Growth

Term	**IEEE 1633 Definition**
Reliability Growth	**(A)** The amount the software reliability improves from operational usage and other stresses. **(B)** The improvement in reliability that results from correction of faults.

Development Environment

Next we will cover best practices for implementing a software reliability program for the IT systems development environment.

Finding and Removing Defects

An important aspect of a software reliability program is the capability to efficiently find and remove defects during the systems development process. The formal testing process (unit, integration, system, acceptance, regression) plays a key role in finding defects as well as verifying that the defects are fixed correctly. The capability to design tests that efficiently find defects is especially helpful when there is a need to test a deterministic process with a binary (no fault/fault) output (response) that is dependent on a large number of discrete inputs (factors) where each input may have many valid settings (levels). In this case it is frequently impossible to execute all of the possible test cases (exhaustive testing) due to the large number of combinations in the factor space. Fortunately, empirical studies by the National Institute of Standards and Technology (NIST) have shown that a very high percentage of software faults arise from the interaction of a small number of factors (typically six or less). This phenomenon makes the use of Combinatorial Optimization (CO) for software testing highly beneficial. CO is an advanced mathematical technique that identifies an efficient number of test cases that effectively covers potentially enormous factor spaces. Using CO tools, such as the NIST's Automated Combinatorial Testing for Software (ACTS), enables the software tester to design a test set for a large test space that is proven to find a high percentage of the existing faults with a test set much smaller than the exhaustive testing approach. ACTS is a proven method for more effective software testing at lower cost. For more information on CO designs, see the STAT COE web site's repository of best practices at www.afit.edu/stat (Bush et al, 2014).

Despite the widespread usage of formal software testing for finding and removing defects, research indicates that pre-test defect removal activities such as inspection of requirements, design, code, user manual, and other system documentation; static analysis of code; informal peer reviews; desk checking; and pair programming are essential to achieving high quality software. Jones found that most forms of testing have a DRE of less than 50% but formal design and code inspections have a DRE of 65% and often top 85% (Jones, 2011). He concluded that improving U.S. average DRE of 85% to acceptable DRE levels of 95% and beyond required pre-test inspections and static analysis (Jones, 2011). Thus, a best practice for software reliability programs is to augment formal testing with an array of high-payoff pre-test defect removal activities.

Software Reliability Working Group (SRWG)

Before entry into the development environment, a process must be created to efficiently and effectively capture and process failure data. A SRWG with both government and vendor membership should be formed to begin collecting and reporting reliability data at the occurrence of a specified event. Requesting reliability data as early as possible in the program motivates the SRWG to create and exercise the process so it is mature before key test activities commence. Establishing a proactive SRWG in preparation for entering the development environment centralizes the implementation of a software reliability program, increasing the likelihood that software reliability issues are given the attention they deserve.

Reliability Data Collection and Management

The foundation of capturing and processing failure data should be a failure reporting, analysis, and corrective action system (FRACAS) – a system, sometimes carried out using automated software management tools, such as the web-based ReliaSoft's XFRACAS. A FRACAS provides a process for reporting, classifying, and analyzing failures, and planning corrective actions in response to those failures. It is typically used in an industrial environment to collect data and record and analyze system failures. A FRACAS system may attempt to manage multiple failure reports and produces a history of failure and corrective actions. FRACAS records the problems related to a product or process and their associated root causes and failure analyses to assist in identifying and implementing corrective actions.

Implementing the FRACAS method promotes the reliability of the software development process by establishing a formal process followed by the entire organization. Some specific benefits of FRACAS include (1) identification of defect patterns, (2) failure data for reliability analysis and (3) central location for lessons learned.

Operational Environment

Finally, we will address best practices for preparing the software reliability program for the IT systems operational environment.

Need for Understanding the Operational Environment

The software reliability program must have a clear understanding of the stresses the system will experience when it goes into the operational environment in order to intelligently assess and ensure its software reliability. First, knowing the magnitude and timing of the stresses placed on the system enables the program team to plan to make available the necessary computational resources at the right time, thereby ensuring the software can reliably provide the requested services. Second, detailed knowledge of the usage patterns of specific transactions allows the program's software reliability team to accurately assess the relative impact of failures caused by each software defect. This knowledge enables the program management team to optimally allocate limited resources to fixing software defects, thereby improving system reliability and user experience.

Developing Operational Profiles

This understanding of system stresses requires careful study of how the users will operate the system and how that use varies over time. Each use, often called a use case, a business process, or a transaction, is a system operation initiated by a user to obtain a desired result (for example, a report is printed, a payment is deposited, etc.). Users will be initiating these transactions according to the mission needs in the operational environment. The program must examine the needs of each type of user to perform each type of transaction to develop the operational profile of the system for the period of interest (Musa 1990). The operational profile of a system is an estimate of the expected number of transactions of each type that will occur during the period of interest (peak daily, average daily, peak weekly, etc.). Table 9 is a simple example of operational profile data for an IT system that shows for each transaction type both the expected mean number of peak daily transactions as well as the expected spread (standard deviation) of peak daily transactions.

Table 9: Example System Operational Profile Data

Transaction Type (ID)	User/Organization	Service Category	Mean Number of Peak Daily Transactions	Standard Deviation of Peak Daily Transactions
1	Customer	Invoice	54.1	1.9
2	Interface, Inc.	Report	12.2	0.8
3	Interface, Inc.	Budget	5.0	0.1
4	ABC (gov't agency)	Report	23.8	1.3
5	ABC (gov't agency)	Budget	9.4	0.8

The first step in developing operational profiles for the system is to identify those transactions that require an operational profile to be developed. Obviously there is no universal approach to doing this step but some questions that may be helpful in identifying transaction types that warrant operational profiles are listed below:

- Are there performance requirements associated with a transaction type?
- Are there high-level transaction types, such as queries and reports, which have specific lower-level sub-transactions, such as a query against multiple tables, which are of particular interest?
- Are a relatively large number of a transaction type expected to occur?
- Is there a high priority associated with a transaction type?

One approach to identifying transaction types would be to simply enumerate all possible transaction types. For example, a transaction could be categorized by user organization (6 interface possibilities), action (new record creation or database query), and subject (employee, contract, or budget). In this example there would be 6x2x3 = 36 unique transaction types.

The second step is to estimate of the number of transactions of each transaction type that occur over the period of interest. For example, the program may be concerned that total system demands in a single hour could degrade the performance to unsatisfactory levels. Thus, we are interested in estimating the maximum number of each transaction type that may occur in a single hour over the course of a day. A simple approach to developing this estimate is to parse every hour of historical data available to locate the hour with the most transactions. Then count the number of each of the transaction types during that hour. The result is the operational profile to be loaded on the system during performance testing to ensure the performance requirements are met even under the most severe conditions expected.

For systems that are replacing legacy systems, historical data may be available to facilitate this task. If historical data and the resources to perform the analysis are available, the STAT COE recommends a more thorough analysis to gain a deeper understanding of what the system will be asked to do. The resulting quantification of uncertainties in the operational environment enables informed, defensible decisions in the tradeoff between risk and test resources.

Conclusion

This paper demonstrated the value of implementing a pragmatic software reliability program for a federal government IT system using the following approaches:

- Establishing common definitions of key software reliability elements based on authoritative sources
- Implementing software reliability metrics and tracking processes that support effective, responsive decision making
- Supporting a software development environment with software reliability enhancements
    - Efficiently finding and removing software defects
    - Software Reliability Working Group
    - Failure Reporting, Analysis, and Corrective Action System
- Developing an operational environment with user profiles that help assess the
    - Impact of stress on system performance
    - Relative impact of each transaction failure on system reliability

References

"Automated Combinatorial Testing for Software" Computer Security Resource Center, https://csrc.nist.gov/Projects/Automated-Combinatorial-Testing-for-Software.

Bush, Brett, et al. "Combinatorial Test Designs." Scientific Test and Analysis Techniques Center of Excellence (STAT COE), 25 March 2014.

"Defense Acquisition Guidebook" Chapter 6 Information Technology & Business Systems, https://www.dau.mil/tools/dag/Pages/DAG-Page-Viewer.aspx?source=https://www.dau.mil/guidebooks/Shared%20Documents%20HTML/Chapter%206%

20Information%20Technology%20and%20Business%20Systems.aspx. (Need to log into Defense Acquisition University web site)

“FRACAS: An Essential Ingredient for Reliability Success” WEIBULL.COM, http://www.weibull.com/hotwire/issue122/relbasics122.htm.

Godbole, Nina S. *Software Quality Assurance: Principles and Practice.* Alpha Science International Ltd, 2004.

“IEEE Standard 1044-2009 - IEEE Standard Classification for Software Anomalies” https://standards.ieee.org/findstds/standard/1044-2009.html.

“IEEE Standard 1633-2016 IEEE Recommended Practice on Software Reliability” https://standards.ieee.org/findstds/standard/1633-2016.html.

Jones, Capers. *Software Defect Removal Efficiency https://www.ppi-int.com/wp-content/uploads/2017/04/Software-Defect-Removal-Efficiency.pdf, 2011.*

Musa, John D., et al. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, 1990.

“XFRACAS” Reliasoft.com, https://www.reliasoft.com/products/reliability-management/xfracas.