Automated Software Testing Implementation Guide

Authored by: Jim Simpson, PhD
Jim Wisnowski, PhD

April 2017

Revised 4 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.

This project is a result of sponsor funding from Office of the Chief of Naval Operations, Innovation Technology Requirements, and Test and Evaluation (OPNAV-N94).

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

Table of Contents

Revision 1, 4 Oct 2018: Formatting and minor typographical/grammatical edits.

Executive Summary

This guide is intended to serve those in the Department of Defense (DoD) interested in applying automation to software testing. It applies a systems engineering process based on the scientific method for the steps to conduct and to achieve an automation capability along with the important need to perform a return on investment (ROI) analysis to make the business case for automation.

Who is the intended reader? Some organizations are considering automation for the first time. For that audience, we recommend executing all phases proposed in the implementation guide. Outside assistance from groups with automation experience is a must. Seek their guidance and benefit from their experiences. In addition, there is a wealth of guidance readily available in texts and online. Some software acquisition programs have had some exposure to, or experience in automation and are interested in improving one or more aspects of their automation process. Common objectives are to select additional system functions or capabilities to automate, to secure more automators, or to change or expand the tools used. For those groups, perhaps only a portion of the guide is applicable.

The guide is organized around the phases of implementation listed below, which are intended to encompass the life cycle of automated software testing within your test program.

- Pre-plan – research, invest time, and gather information for making an informed decision on automation. Perform a cost benefit analysis and compute an ROI, and be sure to include any long-term benefits, then decide whether or not to automate. Specific steps include research into the program test plan, automation capabilities and opportunities, knowing manpower skillset and resource needs, and quantifying costs and benefits from automation.
- Plan – develop an automated software test plan by identifying and prioritizing test requirements, identifying and assessing appropriate automation tools with quantifiable and discernable metrics, identifying barriers to automation implementation, drafting the test automation framework, and outlining the test script needs.
- Design – take the automation plan another level deeper in detail and make decisions for how best to execute automation. In this phase, automation tools are selected and made available, test scenarios for automation are generated, test cases are determined, the output analysis strategy is designed, and configuration control is established, all culminating in a design review.
- Execute – the activities and decisions that enable a test, otherwise to be conducted manually, to be automated. It often starts by interviewing a system operator or capturing the manual tester's steps, then decide the best automated test environment, integrate the tools within the designed test framework, develop and refine the automation scripts, and iteratively test out the execution while refining the process.
- Analyze – the focus is on the output of each automated test, typically involving recording data files or log files. The purpose is to combine, manipulate, and analyze the output data to learn output errors and faults associated with the system under test (SUT) to include integration issues. Individual steps include setting the data format, assessing the output data, ensuring

anomalies are real and characterizing anomalies, revising automation metrics, and ROI.

- Maintain – this final phase is often the most time consuming and painful aspect of automation. Once test scripts have been written, executed, and refined for optimal use, something (SUT, the test environment including monitors or operating system versions and IT patches, automation tool version, etc.) changes. The scripts now fail to execute properly unless revised, which is one of the tasks in the maintenance phase.

Although the phases can be visualized and enacted in a chronological or linear fashion, we realize and stress that there is significant connectivity between them such that moving in a less structured or iterative direction can be advisable. The suggested approach also involves maturing several important automation tasks across multiple phases. For example, automation tool selection is often considered a primary and critical decision. This guide suggests tool selection and tool acquisition be a part of each of the plan, design, and execute phases, where increased knowledge and topic maturity is obtained in subsequent phases. Iteration and looping of the phases is a key to success.

If time is restricted and an automation decision must be made quickly, along with short-turn preparation for automation, be sure to at least consider the following:

- Research automation opportunities and learn which automation tools are best. Know the costs.
- Obtain leadership support by presenting the ROI and quickly identify the barriers to success.
- Learn which parts of your testing are best for automation, design the automation framework, and determine the suite of tools needed for your automation program.
- Find, hire, or grow the automation expertise. Growing can be easier than you think.
- Start with simple automation tasks and increase complexity as automation capability matures.
- Think carefully about how best to cover the input test space and how to get the most of the output from automated tests.
- Decide the automation frequency based on the software development cycle and testing needs.
- Understand that maintenance can be most costly and require the most resources.

Use of the Guide

A useful feature of the Implementation Guide is the frequent placement of summaries, or *Bottom Lines*, throughout to highlight various activities and concisely capture the essence of a step within a phase. One can use the find function within the document to quickly locate all the bottom lines, which can serve as a brief synopsis of the vital tasks to undertake when planning for or conducting software test automation.

Going forward, the intent is to distribute this guide widely and solicit feedback so that the guide can be continually improved. New revisions will be made available, and will eventually be posted on an automated software testing (AST) knowledge center website. It is also hoped that this guide and others like it (e.g. AST Practices and Pitfalls) may be of service to the AST community. Ultimately, we desire to see improved communication and better collaboration among AST professionals and to connect like-

minded people, projects, and interests.

The material herein comes not only from published material including peer-review journal articles, conference material, and textbooks, but even more so from direct conversation with those of you in the DoD taking advantage of automation for testing software intensive systems. The Scientific Test and Analysis Techniques Center of Excellence (STAT COE) is available to assist you as needed and can put you in touch with groups or experts willing to assist as you move towards automated software testing.

Introduction

Automated Software Testing (AST) has had significant impact across the Department of Defense (DoD) and industry. The DoD has not taken full advantage of the efficiencies and improved performance possible using automated approaches across the software development lifecycle. The failure to take full advantage of AST methods is connected to the DoD's lack of understanding of the AST process. The purpose of this Implementation Guide is to provide management and practitioners a handbook that outlines a reusable framework to employ AST methods across a variety of DoD systems.

This effort is part of the Office of the Secretary of Defense (OSD) Deputy Assistant Secretary of Defense, Developmental Test and Evaluation (DASD(DT&E)) STAT COE initiative to better educate programs on the benefits of automated test. The purpose of this manual is to describe the general flow of an AST program from end-to-end and provide insights to activities that will lead to a successful automation effort. The guide describes detailed tasks within the six primary AST phases: pre-plan, plan, design, execute, analyze, and maintain. These phases allow programs to comply with the DoDI 5000.02 (Enclosure 4, paragraph 5.a.(12)) requirements for a test automation strategy. These phases are not necessarily sequential as Figure 1 shows. The activities for successful AST require an iterative approach.

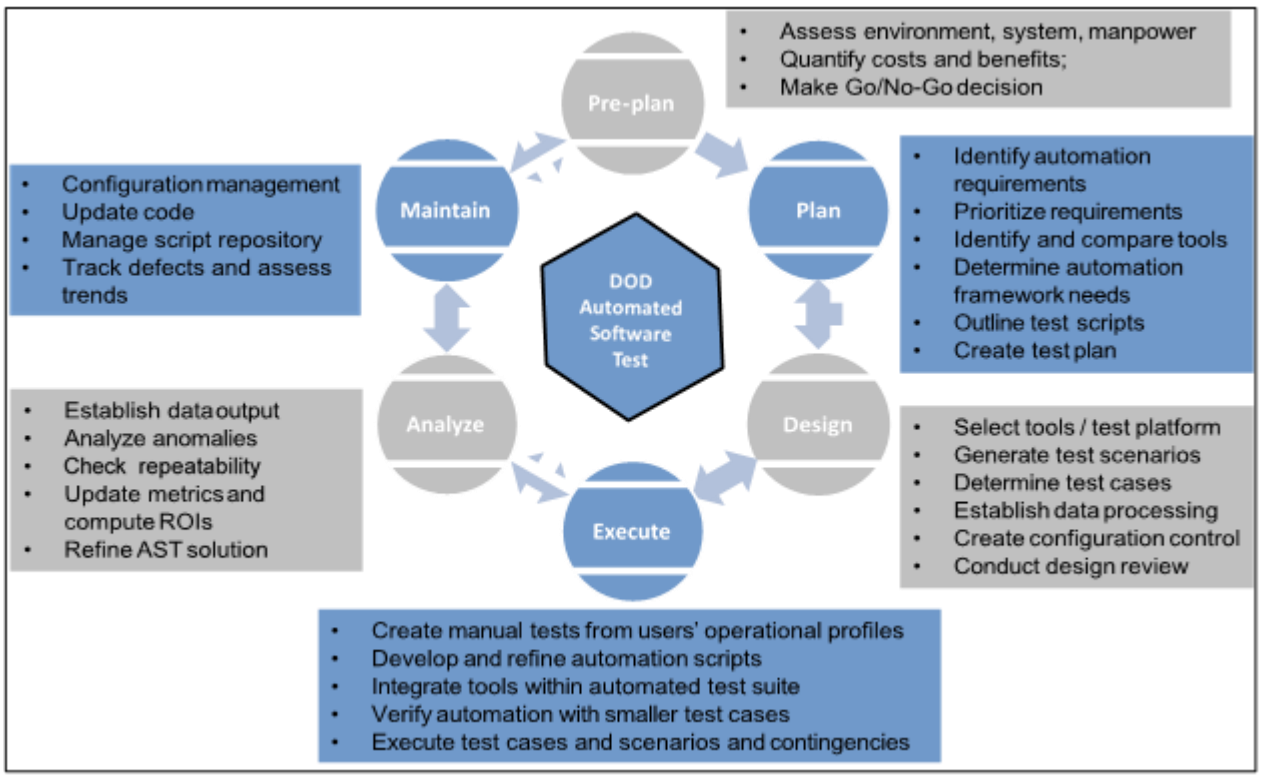

Figure 1. Major phases and tasks for AST programs

The intended audience is leadership (both program and test), system engineers, software engineers, software developers, software testers, and test automators. The tasks are described at a general level and technical details are explained from the vantage point of someone with little knowledge of software test and automation. The AST process flow was developed primarily from interviews with experts across DoD and industry who have had success and failure automating test cases. Additional sources include previous DoD studies, textbooks, technical journals, websites, blogs, and conference briefings. Though every program has unique experiences in the automation journey, there are many common elements that have formed the basis for this recommended methodology for DoD systems.

Phase 0: Pre-Plan

Automation can offer huge improvements in test efficiency and effectiveness but may require substantial investment. Not all programs and requirements should be automated. Before launching into an automated software test program, take the time to assess its value relative to the costs not only in the short term, but especially in the long term. This ROI or business case analysis does not require highly accurate and precise estimates, but does require a sound systems engineering and decision analysis approach. It is essential to have either an operations analyst with these skills and experience or to seek outside assistance or counsel. A rough order of magnitude (ROM) ROI estimate is needed in this pre-planning phase in order to make a 'Go/No-Go' decision to pursue automation.

The Pre-Planning phase will require some time; studying the facets of automation and learning about automation efforts performed on similar or related systems to better understand how the automated testing landscape applies to the specific system requirements. It will go considerably smoother if you have personnel familiar with not only software testing but also automated software testing. If you do not have these resources, fortunately there are organizations across the DoD enterprise that are willing to help – often at no cost. This investment in understanding AST and how it pertains to your system will be beneficial for the project team, particularly if the decision is to automate, but even if automation is not the recommended direction.

Conduct Research on Test Program and Automated Software Test

The first step to determining if an automated approach makes sense is to take the goals, objectives, and requirements for the system under test and look for logical opportunities to apply automation. If you are in early unit testing, say developing software for components of a complex system, tests that would need to be executed only once may not make sense to automate. Conversely, if the system undergoing software development is further along, say undergoing integration testing with new capabilities added repetitively, automation may be tremendously helpful, especially for regression testing to ensure core functionality has not been impacted with the new updates. Many systems will not have access to the software code so only black-box testing will be possible. Whether capable of white, gray, or black-box testing, not all software requirements are testable and of those that can be tested, not all should be automated. Realize that the program documentation (e.g. Capabilities Development Document or System Requirements Document) of system and design requirements/specifications are key inputs, but many other requirements exist and need to be tested based on the expected operational use and operational environment.

You should have team members with some familiarity with AST or at least have reasonable access to individuals with these skills. Some useful resources for a background in AST include the Scientific Test and Analysis Techniques Center of Excellence (https://www.afit.edu/stat/), local organizations with AST expertise such as the SPAWAR Rapid Integration and Test Environment (RITE), commercial vendors (e.g., http://www.idtus.com/; https://smartbear.com/; http://www8.hp.com/us/en/software-solutions/unified-functional-automated-testing/index.html), industry experts, and textbooks such as *Implementing Automated Software Testing* (Dustin et al.), *Experiences of Test Automation* (Graham and Fewster), *Introduction to Software Testing* (Ammann & Offutt), *Foundations of Software Testing* (Mathur), and *Software Testing* (Hambling). Know there is a difference between automation and testing; automation supports testing. The goal is to understand AST at a high level, determine what types of testing can be successfully automated, and generally recognize the value of applying automation.

An important aspect of your research is investigating what automation efforts have occurred (or purposefully not occurred) on relevant systems, whether applied to previous versions of the system, or subsystems, or to similar systems. Here it is essential to investigate broadly and aggressively across organizations and across services. It is not uncommon for software development or acquisition teams (especially in larger programs and in the joint environment) to not have visibility into automation efforts

in closely related programs. Where possible, try to leverage the previous and current automation work (tools and scripts) to quickly gain as much understanding as possible in order to inform the business decision to automate.

The system software development contractor may be conducting AST internally either as a standard practice or contractual requirement. Try to get visibility into what they have done or are doing and whether you can obtain access to their automation work either through deliverables (i.e., Contract Data Requirement Lists or CDRLs) or by a site visit involving demonstrations and documentation.

Bottom line: Conduct research into automation opportunities for your program. Find the right people that can perform a technical assessment, learn about ongoing automation for similar programs, and find out what the contractor is doing with automation.

Understand Your Environment

Automation efforts have a better chance of success when there is not only sufficient capability but also support across the organization. Automation in testing is not the typical mindset of most DoD organizations. One theme across all organizations, government or commercial, is the essential role of leadership championing AST and actively managing the process. Culture may be an important and possibly insurmountable obstacle. If manual testing is the "way we always do it," then a combination of leadership, policy, and technical skill insertion is needed to move forward. AST is the cornerstone as programs transition to a test-driven development (TDD) method with Agile and DevOps. Agile and DevOps also require culture change as software developers embrace change, operators desire stability, and testers focus on risk reduction.

There should be a general understanding of the AST resources required in the current test environment. Examples include personnel (testers and automators), software/tools, host computers, network environment, cloud support, information assurance/cyber protection, software approval processes, enterprise licensing, and so forth. It is essential to understand the overall schedule and flexibility because it may not be possible to automate within the timelines.

Bottom line: Pave the way to automation success by securing leadership support, identifying the major pieces that must be in place in order to automate and then comparing the needed resources to your program's current state. Gain a sense for the hurdles to overcome in order to build an automation capability.

Assess Manpower and Skillset Requirements

There often is a distinct difference in skillsets and experience between software testers and automators. With today's tools, testers with little software development experience can effectively learn to automate some aspects of testing otherwise done manually. However, a robust, streamlined, maintainable, and reusable automation solution often requires significant investment in software coding and development to grow fully independent and reusable automation test scripts that take full advantage of automation capabilities. Rarely will a single automation tool with a friendly graphical user

interface (GUI) be the sole solution for all the automation needs. A suite of tools, each for a different purpose (e.g., browser apps, tracking, unit testing, and continuous testing) with different capabilities integrated into an automation framework, is often the recommended solution. Many tool vendors market their products as not requiring any software development experience, but they succeed in many ways at automating only specific types of tests and have limitations on maintainability and reusability.

There will be significant lead time to hire and/or train personnel to achieve initial automation capability. The time and resources will be a function of automation goals. Management has to decide whether to grow the current workforce internally or hire out the positions. For hiring new staff, they must identify where the talent resides, the best source (military, civil service, or contractor) and then quickly attract the right people, understanding the delays associated with the hiring process. If the decision is to build an AST capability by training the current workforce, it may be difficult to find the individuals with the right potential and motivation who can be freed up from their current duties enough to train and be successful. If this approach is taken, a deliberate training plan that identifies appropriate peers, mentors, coaches, and training timelines is crucial to success.

The long-term benefit to the software acquisition program could be substantial or perhaps an unwise investment depending on how much automation is value added, given the constraints of the system requirements that are testable, the timelines, and expected future efforts. An alternative may be to contract out the work to an experienced AST group, whether government, contractor, or commercial. A detailed discussion of specific knowledge, skills, and abilities is provided in Phase 2: Design for Automation.

Bottom line: Finding or training automators is the single most important investment by any group interested in successful automation. This aspect of the automation process also tends to take the most time and can be expensive depending on the route chosen. Consider all your options in obtaining the right number and experience levels for the project.

Determine Potential Benefits from Automation

The primary goal of automated software testing is to discover defects and opportunities for improved performance more quickly and thoroughly than otherwise would have been achieved using a manual approach. Some metrics to consider when comparing fully manual versus some degree of automation are:

- Increased coverage for lines of code tested
- Increased coverage of expected operational paths and use cases
- Manpower savings over manual testing, especially for repetitive testing (e.g. regression)
- Ability to scale with multiple users and environments; perform high-load and boundary testing
- Ability of test force to focus energies on high priority/high risk areas
- Better output data for analysis and reporting
- Higher defect discovery rate through better coverage and/or freeing manual testing resources for deeper exploratory testing

- Greater delivered software quality
- Shorter time to field system
- Continuous testing to include overnight and weekends
- Reusability of automated scripts

Bottom line: Now is the time to start building a spreadsheet for comparing various automation alternatives. Possible choices are: a) no automation, b) partial or phased automation capability, and c) complete automation where appropriate. Consider the short term and long term impacts of each alternative and use quantifiable metrics such as the ones listed above.

Quantify Costs of the Automation Effort

There are both direct and indirect costs associated with an automation project. Representative direct costs include: software licensing and training; hardware and middleware components for the automated test framework; cloud and network services; labor for learning tools, developing scripts, integrating components, executing tests, and analyzing results; and contractor consulting costs.

Indirect costs can be thought of as the hidden or unexpected time required to automate. Examples are the time taken away from the manual testing function, time to collaborate across many functions, time to keep management informed, time to achieve competence in using the suite of tools, and time to maintain the automated solution. Often overlooked are the maintenance costs required due to frequently changing system configuration and images.

Costs should also be viewed through the short-term versus long-term lens. The direct sunk cost of licensing, tools, training, etc., should be amortized over the expected duration of the automated software test program.

Bottom line: Begin to gather direct and indirect costs associated with gaining an automation capability; both financial and time commitments. Refine estimates as you gather information and learn from experts, and use this knowledge to inform decisions.

Decide Based on Expected Return on Investment

All requirements should not be automated. If there are some tests that only occur once or are on stable software builds and environments, then the marginal benefits would not outweigh the costs. Take time to thoughtfully consider both the quantitative and the qualitative costs and benefits of the automation effort.

It is helpful to estimate the ratio of the expected additional time to develop an automated routine to the time it takes to run a manual test. As mentioned in the benefits and costs sections, many of the actual benefits and costs are not considered up front. It would be prudent to make this evaluation across a range of expected costs and benefits (i.e., optimistic, realistic, pessimistic) to see the sensitivity of the results.

Leadership has to use the analyses to balance the assumptions, expectations, and program pressures to

assess risk. Ultimately, they have to make a 'Go/No-Go' decision as early as possible; but not before they have been provided a solid business case from the staff. Typical leadership roles making these decisions would be the Integrated Product Team Test Lead, Director of Engineering, Program Manager, and Software leadership.

If an AST program fails to provide sufficient ROI, carefully consider whether this project could serve as a stepping stone for a much greater capability the next time. Many costs have already been realized and lessons learned can make the next AST program much more worthwhile.

Bottom line: Now is the time to make the decision: automate or not. Be sure to convert the metrics from automation into quantifiable gains (time and money) to the program, and then have your operations analyst compute the automation return on investment for the short term (1 year) and longer term (2-4 years if appropriate to the program). Use the ROI to make your decision and be sure to document all analyses along with minutes of decision briefings.

Phase 1: Plan

Upon deciding to pursue an automated software test approach, it is now time to perform more detailed planning. Fortunately, a fair amount of planning has already occurred to create the business case which can take the form of: research into automation, automation taking place elsewhere, quantifying resources needed, and determining possible tool solutions and costing options. A more deliberate planning approach is needed now in Phase 1, which will leverage the information already learned and, equally as important, build upon the relationships already established from Phase 0. Leadership still has the most important role in managing the dynamic environment as automation may be new territory for the program.

Detailed information is needed on the system requirements that can be automated and how best to prioritize them using a risk management approach. Planning includes establishing an approach to the AST framework that consists of the hardware, network, models, tools, and analysis methods. The Planning Phase also includes running small automated scripts as a proof of concept and capability. These could be as simple as logging on to the system under test and selecting something from the main menu using the selected tools.

Bottom line: In the planning phase, it is time to assess what is now known about the automation road ahead, and to start building upon and working out a detailed action plan for each of the steps from the pre-plan phase. Leadership support is emphasized as well as identifying the aspects of the program to automate and the tools best suited for use.

Plan Phase Deliverable: The output of this phase is a test plan that outlines the expected requirements tested with AST, the overall strategy with representative tasks, the resources required, the responsible point of contact (POC), and plan of action and milestones (POA&M).

Identify Automation Requirements

Requirements are statements of expected functionality. Software requirements have already been addressed in Pre-planning at a high level. We build on that foundation in the Plan phase to focus on the most important requirements offering the best opportunity for successful automation. The goal is to think beyond traditional requirements and the traditional depth/breadth of testing as automation can enable more rigorous and complete testing of the system under test.

Careful test planning always starts by forming concise yet comprehensive objectives, and for most phases of developmental testing, objectives come from the requirements. Software testing requirements will have to be distilled from various sources. Rarely will there be a comprehensive and well-thought out list of requirements with testable criteria. These must evolve through an iterative and collaborative approach where stakeholders can agree to a common set of requirements as well as a prioritization scheme.

Requirements should be traceable to actual system capability and functions. Some source documents include Capabilities Development Documents, Capabilities Production Documents, Concept of Operations, Operational Mode Summary/Mission Profiles, system specifications, system software specification, and component specifications. Other documents could be System Engineering Plans, Test and Evaluation Master Plans (TEMPs), test plans/detailed test procedures of similar systems, and contractor design documents and test plans. The goal is to use the defined requirements and mission threads to develop test designs. A meticulous process decomposition using flow charts and activity diagrams will identify many sub-requirements that trace to a system function. Depending on the phase of test (e.g. unit, integration, functional, performance in Developmental Testing (DT), integrated DT/OT, Operational Testing (OT)), the testing scope in level of detail and number of requirements will vary. It is helpful to have a requirements management (RM) system in place enabled by a tool such as Atlassian JIRA to help plan and track automation requirements. Other common tools are IBM Rational DOORS, qTest, and XQual.

There are other requirements that need to be addressed in the Plan phase. Non-functional requirements such as the expected operating system, information assurance features, hardware systems, and other environmental factors need to be accounted for. The requirements for the AST framework should also start taking shape and will be addressed in subsequent sections.

One critical factor in planning and executing automated testing is whether the automation will be used to generate complete tests, input data and expected results, or input data only. A variety of methods are available to automate the generation of test oracles (see section on Determine Approach to Test Oracle). Most of these rely on formal models in some form, or assertions embedded in code which serve effectively as a partial formal model. Other options include partial correctness approaches such as metamorphic testing and other "sanity checks." In some cases, fuzz testing will be a useful preliminary step, and this should be included in planning discussion as well. References include Barr et al (2015), Bartholomew (2013), Kuhn and Okun (2006).

Bottom line: The places to start in detailed automation planning are the system requirements and intended operational capabilities. Collect all the available system requirements documents and develop a comprehensive test strategy, such that all the opportunities for value-added automation are listed.

Develop DEF-based Requirements Prioritization

MITRE created and DASD (DT&E) has endorsed the Developmental Evaluation Framework (DEF) shown in Figure 2 to provide an analytically tractable way to manage a test program to best support acquisition decisions. This methodology has also been used throughout the DoD and is directly applicable to AST implementation. The DEF supports the requirements decomposition process where the goal is to flow from a capability to precisely defined and testable requirements. The capabilities are the Developmental Evaluation Objectives (DEO), which we can translate to the software testing world as functions. These DEOs are categorized into broad areas such as system performance, cyber-security, interoperability, and reliability & maintainability. These objectives are broken down even further with technical measures assigned to each. The AST process would break down these system requirements and test & evaluation (T&E) measures even further into the functional components and tasks. This extra level of detail should be added to the existing DEF matrix to ensure requirements traceability. These tasks form the basis for test cases and scenario generation. The right side of Figure 2 could be adapted by replacing the 'Decisions' with 'specific scenarios of interest' and the entries would be the applicable test cases.

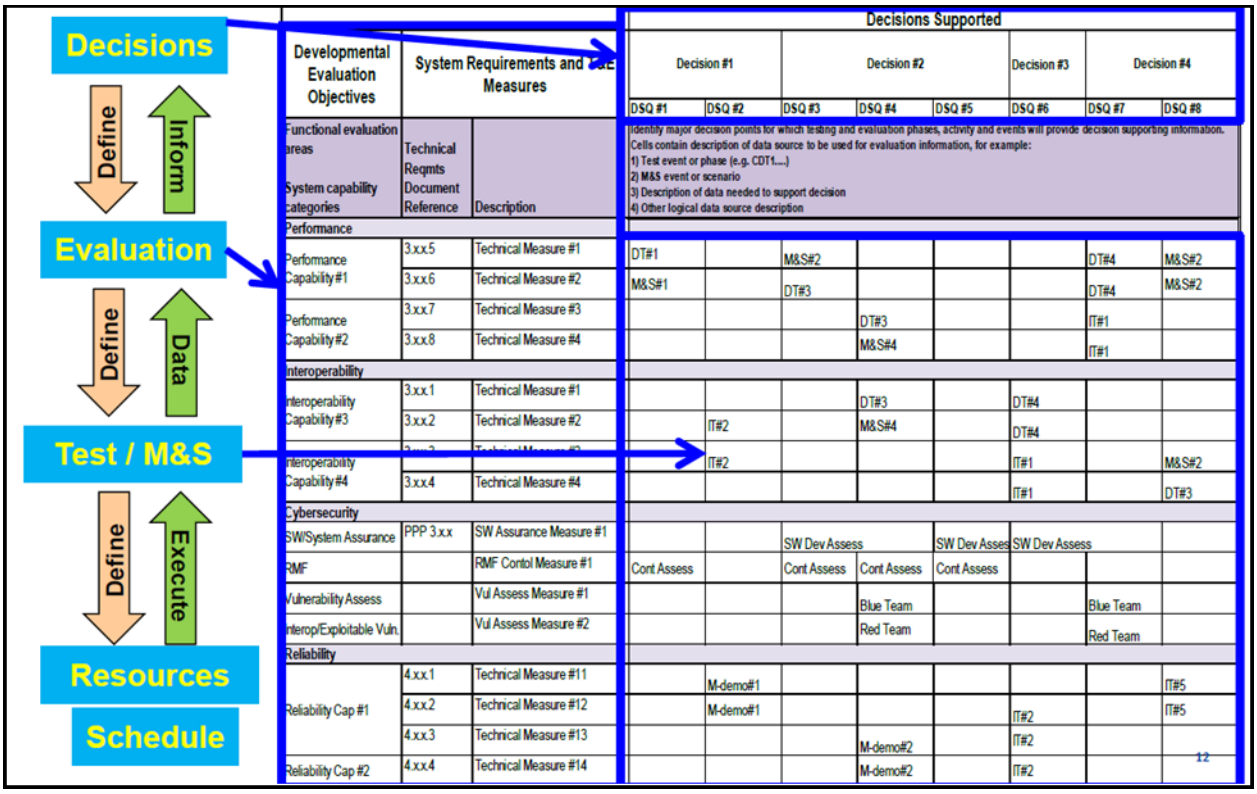

Figure 2. MITRE Developmental Evaluation Framework

The DEF process documents sources for each requirement and can be used to help prioritize which

requirements get automated. The hope is that the most important capabilities receive top priority, but that does not necessarily translate to the correct prioritization of the AST requirements. Additional consideration needs to be given to the risks of automation. Automation can be more complex depending on the requirement and the consequences of failure can be higher for testing some requirements. These value/risk tradeoffs should be evaluated and integrated into an overall prioritization scheme. Attention and weights should be considered for:

- Contribution to DEOs (which are rank ordered)
- Probability that automating the requirement fails based on complexity or script development delays
- Consequence of failing to automate the requirement
- Ability of manual methods to effectively test the requirement

There are many ways to compute a priority number, such as a weighted average. The test team should integrate the prioritized list of requirements in all activities across the AST lifecycle.

Bottom line: Take the previous step's requirements and build a DEF, focusing on automation opportunities. The DEF process decomposition converts the written requirements to testable requirements, allowing the automation team to see which types of tests would be recurring and better suited for automation.

Once the team understands and prioritizes the AST requirements, they may consider involving the system contractor. The team can review the Request for Proposal if possible and include language directing an AST program. Deliverables can include results, reusable scripts, assessment of AST utility, and other information. Additionally, Operational Test Agencies should look at the possible benefits of automation during preparation of the TEMP for Milestone B; they may be part of the development and execution teams.

Identify and Compare Tools

The number and types of tools available for AST can be overwhelming. Each phase of an automated test program potentially has a requirement for a different tool or set of tools that may need to be integrated with a disparate collection of tools. The goal in the Planning Phase is to understand the general capabilities of the relevant tools and down-select to a few promising candidates that will fit your workforce, timing, and desired level of automation rigor. A thorough assessment of needs will result in a multiple tool solution.

Begin with the tools your team uses or feels comfortable with and can efficiently use for the upcoming AST. These tools will have higher priority due to the associated shorter learning curve and better overall automation probability of success. Write up a brief gap analysis of these tools to find the capability holes in order to identify alternative tool solutions. Perform a broad search for possible tools such as those used historically, open source to include tools built specifically for DoD use, freeware, and commercial packages. Take time to interview experts to find out what they are using and why, what they have tried

but no longer use, and what they would like to use if their program was unconstrained. There are multiple repositories of tool evaluations and program usage profiles across the DoD (e.g. SPAWAR AST Tools Database and Gunter AFB AST Tools Database) which could also help identify promising contenders.

Some measures to keep in mind as you consider possible tool solutions are:

- Are the types of tests modular and capable of being shared across application domains?
- Does the tool use a common scripting language like VB, JavaScript, Python, etc.?
- Where in the testing or acquisition lifecycle can it automate?
- What operating systems does it support? Is it for web-based systems only?
- What is the ease of use and how big is the learning curve to effectively automate?
- Can the team be reasonably expected to efficiently use the tool?
- Does it have a GUI capture-replay capability only or does it support application programmer interface (API) calls to the GUI?
- What are the total costs to include licensing, training, maintenance, and support? Are enterprise licenses available elsewhere that can be used?
- How responsive is the vendor for support questions and troubleshooting? What training or user community is available free on-line?
- What are the information assurance hurdles? Can it be integrated into and operated on government computers (e.g. Navy Marine Corps Internet (NMCI)) and if so, how long is the approval process?
- What are the output products and can they be easily accessed and customized? How well does it provide insight for debugging (or fault identification) versus just showing that a test has failed?
- Is there a history of Verification, Validation, and Accreditation (VV&A) within the DoD?

Bottom line: A tools assessment is one of the most time-intensive yet rewarding parts of the planning process. Often, automation projects have selected a tool based on very limited information (e.g. only one they have heard of) and later regret their decision (pay now or pay later syndrome). Better automation tools typically require automators to have the time and motivation to learn the tool, resulting in more stable (less maintenance) and more extensive automation capability. Consider all reasonable options.

Determine Automation Needs

Automated software testing has many definitions and interpretations. Depending on the program and staffing, AST can be quite complex, taking the test team years to develop the right automation framework or it can be quite simple where a few basic steps of a test are automated, like running batch files overnight. There should be a general understanding of how complex the AST will be coming out of Phase 0 research. An important overall consideration is the level of abstraction. That is, how does the system operate during test? It could be live operators, virtual scripts, modeling and simulation, or running a pre-recorded mission thread. This Live, Virtual, Constructive (LVC) environment will influence

many other facets of the testing to include choice of venue (virtual network, software integration lab (SIL), contractor facility, or operational environment).

Based on the automation requirements and potential available tools, determine possible matches to best execute an AST in both the short and long-term. Possible considerations include:

- Bitmap capture-replay that limits how test inputs may be applied and how system responses are evaluated versus the de facto standard (currently) of direct programmatic access to GUI APIs
- Need to maintain system integrity and mimic operational performance by hosting the tool apart from the system under test by not altering its source code (client/slave relationship)
- Planning to ensure convenient reports that show evidence that the test properly executed
- Ability to have data-driven tests with unique data entry flexibility; accounting for the reality of a reduced installation and test time
- Requirements for shorter tool learning curves
- Minimizing sustainment/maintenance time for software version changes and tool version changes
- Ability to conduct both positive and negative testing where faults are intentionally inserted

Bottom line: First, understand the testing environment, current resources for tools, and planned automator capabilities. Form a small team (program management, test lead, software engineer) to perform tools assessment to determine the best, feasible tool solutions.

Outline Test Scripts

Creating test scripts is a continuous process that begins in the planning phase, takes shape in the design phase, becomes operational in the execution phase, and is updated often in the maintenance phase. In planning, the goal is to outline the overall approach to building the test scripts and then create scripts for a few simple tasks.

A critical task in this phase is to query test script repositories to see if there are any that can be used or slightly modified to save on automation development time. There is no formal DoD repository of scripts, but likely testers and automators from earlier or existing AST programs have cataloged and documented their work. Be sure to reach out to all relevant programs and testers because even if they have scripts to immediately apply, they will also have helpful recommendations and lessons learned.

For new scripts – start small and build. Take elements of a representative test case and try automating it with different tools and different ways within the same tool. Experiment with negative testing to see the output reports and fault detection logic.

Once a rudimentary automation capability exists, outline the expected flow for building the test scripts of the full project. Understand there will be many iterations of this catalog but at least begin to think about how to effectively test in an automated environment.

Bottom line: Working on outlining the needs for test scripts and building simple scripts transitions a

general plan into tangible products, bringing automation to life. Obviously, the benefits here are that the automators can see the bigger test script landscape and gain valuable experience with various tools by automating.

Publish Automated Software Test Plan

Documentation is important to ensure everyone (management, test leads, software developers, system engineers, software engineers, automators, etc.) understands and agrees with the overall AST approach.

Bottom line: The deliverable out of Phase 1 is an Automated Software Test Plan. The test plan is a living document and should clearly articulate the planned approach. Sections should include but not be limited to:

- Detailed system description
- Delineated requirements along with the DEF decomposition matrix down to automatable tasks
- Prioritization methodology and ranking
- Planned AST framework
- Candidate tools along with capability evaluations
- Responsible POCs for critical tasks
- Test resources
- Timelines

Phase 2: Design for Automation

The Test Plan provides a strong foundation and preparation for the design phase. The purpose of this phase is to provide more detail into the automated software test plan and make important decisions that will shape how the tests will actually be executed. The activities in the design phase do not occur linearly but are worked on simultaneously and iteratively. As the design matures, assumptions are revisited and different paths may result. The entire design phase should be accomplished in a highly collaborative environment as expertise across many functional areas is required to develop a coherent AST design.

Design Phase Deliverable: The deliverable out of this phase is the content for and execution of a detailed design review. This review should be a natural extension to the collaboration across the automation team that is taking place all along, characterized by extensive peer review, walk-throughs, and strategy sessions. The design review should result in a recommendation to proceed to automated test execution. Alternatively, the team may need to focus more energy on certain aspects of the design to ensure successful automation or the collective group may determine the additional time spent understanding the system and developing the AST has educated them enough to decide to stop the automation.

Determine What to Automate

After completing the requirements definition, the development evaluation framework, tool research, and consultation with test automation experts, it should be clear which tests are the most promising to

automate. Decisions regarding what can and should be automated will be influenced by the software development life cycle (unit test/development, integration, functional, and performance) and test phases (DT, integrated DT/OT, and OT). Factor in ease of automation, success of automating like requirements across other programs, the team's expected capability level, the potential improvement in test coverage, and how often a test will need to be run manually. Balance these considerations against the risks associated with upfront fixed automation investment costs, penalties for scheduling delays due specifically to automation challenges, additional workforce development, and initial time investment taken away from manual testing.

It is helpful to bin automation opportunities into initial categories of easy, moderate, difficult, and very difficult, and to sequentially automate in this fashion. This ordering will allow the team to gradually sharpen their skills while achieving success along the way. Multiple automators will benefit from each other's learning, which accelerates automation capabilities.

Bottom line: Now decide where to automate. The entire automation set of tasks should be laid out, then sequenced based on skills of the automators, complexity of the automations, and automation priorities.

Select Tools

Phase 1 identified a candidate set of tools applicable to the expected AST. As the requirements and test environment have become more mature, the preferred toolset should become more obvious. Select the automated test tools appropriate for the requirements. There will be additional research required to make the best selection for capability and usability. Sources include guidebooks, texts, related studies, other DoD AST efforts' tools, vendor sites, and demonstrations. It is likely tools not initially considered will now enter the mix. There are also the more popular DoD AST functional testing tools that have widespread support and subject matter experts (SMEs) throughout the department: Innovative Defense Technology's (IDT) Automated Test and ReTest (ATRT), Hewlett Packard's Unified Functional Testing (HP UFT), SmartBear's TestComplete, Microsoft's Team Foundation Server, and open source Selenium.

Other helpful information is understanding how all of the seemingly disparate organizations may have synergies with the possible existence of enterprise licenses or qualify for reduced acquisition costs through combining seats. Seek available expertise such as the team running the tools on the Hanscom MILCLOUD or the automation SMEs at SPAWAR's RITE group. Once the tools are narrowed down, learn them. Practice with easy test cases, on-line tutorials, and vendor documentation to become proficient as quickly as possible.

The suite of tools will not necessarily seamlessly work with one another—there can be substantial integration issues that may require clever coding solutions, so think about an automation framework. A skilled software coder or outsourcing this function may be necessary to get the entire suite to function properly.

Bottom line: Take the tool decision process from the planning phase, and acquire the tools chosen. Be sure to look for enterprise licenses, or to see about joining other groups for volume discounts. Open source tools offer advantages as well. Also, ensure the tool can be loaded on government computers

and the tools can work together.

Build Automation Capability

An AST-enabled test team has software engineers, test engineers, software testers, and automated software testers. A high-performance team is created either by training and developing current testers/engineers into automators or by hiring personnel (full or part-time) with those skills. The decision on how to achieve automation capability is a function primarily of the automation requirements, resources available (time and funding profile), and the in-place team's skillset as well as potential for growth. To grow an in-house automation capability, the team needs time for formal training, self-teaching, networking, and mentoring. Instant expertise may be available by "borrowing" neighboring automators, contracting out consultants, or hiring full-time equivalents; whether military, civilian, or contractors.

Bottom line: Recruit your automators internally or externally. Only a few are needed to be successful, but they need to be capable of, and committed to automation. The educational or technical background of the individual is not as important as the passion to learn the ability to automate.

Determine Automated Test Platform and Framework

A test team enabled with the right AST skillsets and tools needs to be resourced to create an adequate AST framework. A framework can be thought of as the components needed to automate testing the SUT. An example would be a simulation model that stimulates the SUT allowing automated operations to achieve the desired function. Considerations include the proper client/slave configuration, selection of an operating system, networking and potential use of the cloud, accounting for simultaneous user needs, information assurance and classification needs, and hardware requirements.

The framework also includes the suite of AST tools. The tools can control the entire test process and are also used for finding software failure and output analyses. A notional example of a simple framework from Distributed Common Ground Station-Navy Increment 2 (DCGS-N Inc 2) is shown in Figure 3 below.

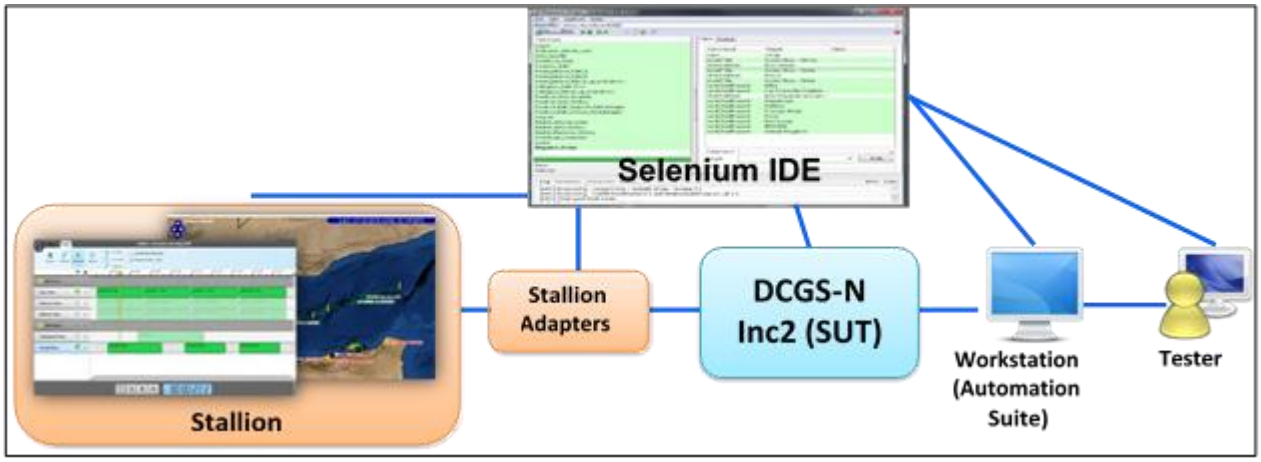

Figure 3. Notational DCGS-N Inc 2 automated test tool integration framework

Generate Scenarios and Test Cases

Test cases are derived from the prioritized requirements matrix and should be "linked" to the requirement to ensure coverage metrics. A test case may incorporate several requirements and multiple test cases define a scenario. Beware of requirements that are tested in multiple test cases or not at all. The scenario is a required function or capability of the SUT. The test cases should be simple steps that are easy to understand and may require prerequisite test cases. Test cases should be easy to update as requirements change and also should be well documented. Consider designing 'smoke tests' to ensure critical functionality is exercised by grouping several test cases together.

There are many scenario generation tools that should have been vetted in the Tool Selection task. These can help; but, most of the work will be manual in a collaborative setting with SMEs trying to map the requirements to reasonable test cases. Carefully consider the new automated testing paradigm, which may allow for broadening the test scenario coverage. Decide what data feeds are required, the method of fusion, and approach.

Bottom line: Query the system experts or look to historical testing and develop test case scenarios. This aspect of the process often consists of an experienced system operator or manual tester sitting down with the automator and automating a manual test case using record and playback. Iterate on the automation to improve the automation and make it robust to changes in the software configurations.

Determine Test Case Coverage

Coverage can refer to the percentage of lines of code of the SUT that are exercised in a test program. It can also measure the proportion of requirements tested in a scenario or across test cases. More often, coverage relates to the number or percentage of use cases or possible paths that are tested out of all possible combinations.

Target adequate coverage, and decide on the correct approach for maximizing the coverage of the software operating conditions. Spend time devising the factors that characterize the SUT's expected functionality in order to build the test design. Possible strategies and methods include:

- Risk-based testing that focuses on critical performance areas with high probability of failure
- Statistically-based test matrices from the design of experiments discipline (e.g. fractional factorials)
- Combinatorial or factor covering arrays that minimize runs required to test a specified order of interaction referred to as strength
- Distance-based or space-filling designs used with continuous factors to uniformly populate the design space

After designing test cases, determine the percentage of coverage using the appropriate metrics for the given methodology. Note that the desired coverage could have a substantial impact on costs and should be weighed appropriately as part of the overall risk management framework.

There may also be contractual or regulatory requirements for particular coverage criteria to be

achieved. For example, there are test criteria required by the Federal Aviation Administration (FAA) for civil aviation. Avionics and other equipment used in DoD systems may also be used for commercial application, so the software would be required to meet FAA requirements.

A variety of tools are available for measuring structural and lines of code coverage. Many are open source (e.g. gcov, Code Cover, JavaCodeCoverage, JFeature) while others are commercial (e.g. Clover, BullseyeCoverage, TestCocoon, eXVantage).

Decide Test Timing

One advantage of AST is that testing can be executed without much human intervention. Tests can be run during off hours, during the weekend, or in the background while testers work on other projects. This flexibility would have been first considered during the 'Go/No Go' process. Based on this flexibility, teams must decide on the practical and desired frequency of testing to be performed. Are tests to be performed after agile sprints (monthly), capability deliveries (quarterly), or overnight during the sprints as Command Run On (CRON) jobs?

The flexibility of test timing can be affected by the duration of the individual test cases and the allotted time. The duration of a test is often dictated by the test case complexity and the performance of the chosen AST framework. The tools vary dramatically in how fast the automated scripts run. GUI-based tools or scripts tend to run near real time, while some object or code-based scripts can reduce execution times by more than 90-percent from real time. If a test case can be run at various depths and the tester has determined the execution times for each depth of test, the tester can control the duration of a test case by selecting the appropriate depth of test. Thus, the tester must consider the amount of control over test case execution times and the length of the allotted test time window when scheduling individual test cases to achieve the chosen objective (maximize coverage, etc.).

Bottom line: Determine the frequency of testing based on the software development cycle and testing needs.

Establish Data Acquisition and Processing Systems

Determine the needs and best approach for collecting the data output from the automated testing. Set the system up to capture the data which may be as simple as a spreadsheet. Consider running prototype tests with negative testing (inducing faults) to identify areas where greater detail is required for log and output files. Focus efforts first on the ease of identifying where the software error occurred and provide any relevant information that could be useful in a root cause analysis. Decide how best to reduce, analyze, visualize and post-process the data to maximize insight into SUT performance.

Determine Approach to Test Oracle

The test oracle is the process of determining whether or not a test has passed or failed. Even with automated methods of generating input data and running tests, the oracle problem remains. Testing requires both test data and results that should be expected for each data input. This is generally the costliest part of the testing effort, since extensive human involvement is needed in conventional

approaches. However, a variety of methods are available to automate some or all of the test oracle generation. These vary in initial cost, level of sophistication, and domains of application. Automated approaches to solving the oracle problem include:

- ***Fuzz/crash testing*** is typically done by recording some data input, then randomly permuting data fields to generate a large number of tests (possibly tens to hundreds of thousands). These are run to determine if any inputs cause the system to crash or freeze. Fuzz testing is used extensively by commercial software developers (less for functionality) for early detection of major faults with numerous tools and references available.
- ***Embedded assertions*** is a popular "light-weight formal methods" technique to embed assertions within code to ensure proper relationships between data, i.e., as preconditions, post-conditions, or input value checks. Tools such as the Java Modeling Language (JML) can be used to introduce very complex assertions, effectively embedding a formal specification within the code. The embedded assertions serve as an executable form of the specification, thus providing an oracle for the testing phase. Reference: du Bousquet, L., Ledru, Y., Maury, O., Oriat, C. and Lanet, J.L., 2004, September. "A case study in JML-based software validation." In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on* (pp. 294-297). IEEE.
- ***Model based test generation*** is a formal (mathematical) model of the system under test, typically expressed in temporal logic or as an extended state machine, or in design notations such as Unified Modeling Language (UML). A simulator or model checker is then used to generate expected results for each input. If a simulator can be used, expected results can be generated directly from the simulation, but model checkers are widely available and can also be used to prove properties such as liveness in parallel processes in addition to generating tests. Conceptually, a model checker can be viewed as exploring all states of a system model to determine if a property claimed in a specification statement is true. Reference: Bartholomew, R., 2013, May. "An industry proof-of- concept demonstration of automated combinatorial test." In *Automation of Software Test (AST), 2013 8th International Workshop on* (pp. 118-124). IEEE.
- ***Metamorphic testing*** uses a small set of tests whose expected outcome has been determined manually. Then system properties are used to generate other tests with different inputs, whose expected outcomes can be produced from the original test. For example, in testing a sine function, it must be the case that $\sin x = \sin(\pi - x)$. Thus, the program is tested for a correct result for sin *x*, then a new test can be generated using the input $\sin(\pi - x)$. Reference: Liu, H., Kuo, F.C., Towey, D. and Chen, T.Y., 2014. "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, *40*(1), pp. 4-22.
- ***Match testing using two-layer covering arrays*** suggests that test settings for an input factor may represent ranges of values (called equivalence classes) for which the output is expected to remain unchanged. For example, a shipping program may charge the same rate for any package under one pound, a second rate for packages one pound to 10 pounds, and a third rate for packages over 10 pounds. Values within each of these ranges are equivalent with respect to the cost calculation. Any value within an equivalent range may be substituted for any other and the program output should be unchanged. The test method works by generating two test arrays: a

primary array and a secondary array. The entries of the primary array represent names of equivalence classes of input factors. For each test row of the primary array, a second array is computed. The settings in the second array are the values from equivalence classes corresponding to the names of equivalence classes in the primary array. If the outputs corresponding to one row of the primary array differ, then either the equivalence classes were defined incorrectly or the code is faulty in some way. Reference: Kuhn, D.R., Kacker, R.N., Lei, Y. and Torres-Jimenez, J., 2015, April. "Equivalence class verification and oracle-free testing using two-layer covering arrays." In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (pp. 1-4). IEEE.

- ***Classification tree method*** is a graphical method for analyzing program inputs and their value partitions, then turning these into test cases. A tree structure is defined with one branch for each parameter or factor in the program inputs. For each of these, branches are then defined for each equivalence class of that parameter or factor. Equivalence partitioning is done as in other test approaches. After the tree has been constructed, weights can be attached for the frequency of occurrence of factor values in inputs, which are then used in optimizing and prioritizing tests. Reference: Kruse, P.M., 2016, April. "Test oracles and test script generation in combinatorial testing." In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on* (pp. 75-82). IEEE.
- ***Pseudo-exhaustive testing*** relies on the fact that not all outputs depend on every possible combination of input variables. The method depends on exhaustive testing of all combinations of variable values that truly matter using combinatorial arrays along with the automated generation of test oracles for model checking. An advantage of this method is that it can be used to produce a complete test set in the sense that all negative cases as well as all positive cases are verified. Two arrays are generated, one for positive tests and one for negative. Reference: Kuhn, D.R., Hu, V., Ferraiolo, D.F., Kacker, R.N. and Lei, Y., 2016, April. Pseudo-exhaustive testing of attribute based access control rules. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on* (pp. 51-58). IEEE.

Create Configuration Control Construct

Changes to both the SUT and the automation framework are common across DoD applications and need to be tracked. Configuration control is the process for managing and tracking these changes. There are numerous tools available to help an AST program with configuration control including JIRA and Subversion-Source Control which are hosted on the open-source (for DoD) Defense Intelligence Information Enterprise (DI2E). Independent of the tool selected, the team must enforce a disciplined process of keeping the database up-to-date regularly as some test cases may not run correctly on a different version of the SUT. Keep detailed records of changes and assign points of contact to track specific capability areas.

Conduct Design Review

Design Phase Deliverable: Plan and conduct a review presentation and discussion to demonstrate readiness for automation execution. Phase 0: Pre-Plan should include a brief summary of the pre-plan

decision process and evidence for the automation decision. Phase 1: Plan should be covered in detail and highlight any incompletes, surprises (good or bad), and areas of concern. Phase 2: Design, again, should be in detail, and showcase that the primary steps have been taken to ensure readiness to execute automation. Evidence of readiness can include a demonstration of initial capability with smaller prototype scripts. The team should revisit the ROI of the project again before committing the additional resources to enter into Phase 3: Execute.

Phase 3: Execute

The word "execute" in software is associated with the program running. For AST, there is a task where the scripts run to execute the automated test. However, Phase 3 in this methodology refers to the collection of activities that enable a test to become automated. This phase covers the flow from seeing how users employ the system, to manually testing that process, writing scripts to automate it, and finally, running the automated test. There is intentional redundancy within the design phase which sets up this execution phase.

Capture Operator's Application of the System

The test team needs to be familiar with how the SUT is actually used operationally. A SME usually provides the functions most commonly executed, along with the functions that are occasionally performed. The automator can then build the pixel- or code-based script (GUI versus API) that captures the primary process flow for the mission threads, together with all related contingency paths.

The team should also focus not only on the most frequently executed paths, but also on the highest risk ones in terms the SUT's ability to perform intended functions. The team should also make an assessment of the automation potential for these mission threads—just because they are critical does not mean the technical challenge of implementing automated tests is any easier.

Create Manual Tests

First ensure that the manual test processes of the tests expected to be automated are adequately and sufficiently captured. The test automator will participate with the software test engineers in a series of manual tests that are detailed in a step-by-step procedure. The automator will make note of promising constructs to integrate into the automated solution. If possible, other testers should also run the test manually with the automator to check for consistency along with alternative and innovative solutions.

Decide Automated Test Environments

The test team will have to evaluate test execution options, though some will be dictated by the software development lifecycle and maturity of the SUT. There may be no choice on where the test is being conducted: at a software integration lab (SIL); on a cloud server; via the web, Secured Internet Protocol Router Network (SIPRNet) or Joint Worldwide Intelligence Communication System (JWICS); in a controlled operationally representative environment/range; or completely operational environment (e.g. on board a ship or aircraft). There may be multiple ways to stimulate the system in order to see if it works effectively. These levels of abstraction are live, simulated live (virtual), recorded live, and

constructive via modelling and simulation.

Questions to ask include:

- How and where will the software be tested?
- Is the emphasis on developmental or operational test?
- Will the true system operators be involved?
- What are the hardware requirements?
- Are simultaneous users or different load conditions of interest?
- Are alternative classification systems involved?

Integrate Tools within the Automated Test Framework

The selected tools from the design phase should already be part of the framework. Test communication between tools using prototype or simple test scripts. Engage the software developers/coders if there are compatibility issues or ineffective linking and communication among software tools. For example, make sure the open source tools work effectively with commercial parent tools.

As the automated solution and environment matures, often additional tools are needed to achieve desired automation features. This is especially evident as more and more output data is generated and test teams are faced with challenges of how to use it.

Develop and Refine Automation Scripts

The design phase produced draft automation scripts for most of the test cases. In the execution phase, the scripts are finalized through an iterative process in a highly collaborative setting possibly hosted on the cloud. There should be detailed code reviews by peers and continued work with SMEs to adequately verify and validate the automated software test.

There may be considerable refinement necessary for scripts that were “borrowed” from other users or a repository. They may have seemed to be plug-and-play, but rarely will these products require no modification. One common “borrowed” script is for user authentication – a Common Access Card (CAC) – for example. This ordinarily would require the tester to be present; however, simple scripts using open source tools (such as AutoIT) can be developed or acquired to automate this activity.

Verify Automation Pilot Results

Output from Phase 2 is an estimate of the difficulty to automate proposed test cases. Simpler test cases should be automated first to give the AST program the best chance of succeeding. In a collaborative environment, walk through the results output to the log file. These outputs can be screen shots or messages indicating success or failure in executing a step.

It is very helpful to perform negative testing of the automation to ensure the automation catches the input errors. Negative testing will also help the team evaluate whether the output can be effectively interpreted manually or if it is written to the correct files using proper formatting. There may be modifications to the messaging architecture required to better understand the output.

Execute the Automation

This execution task refers to the actual process of running the AST program. The goal is to execute the AST program without test team intervention either off hours or in the background, not interfering with other test activities. There still may be a need to monitor the AST as errors may occur in the SUT stopping the automation early, or there may be errors in the automation framework. It is also not uncommon for some tools to get "hung up" looking for the correct image in order to proceed to a subsequent step. Be aware of automation that requires interim manual inputs; here, the testing is not fully autonomous.

The team should look for opportunities to correct or improve the automation. Identify the faults that stop the automation process and try to make the framework as robust as possible. Promptly correct scripts that execute incorrect logic and those scripts that fail to fully execute the intended functions.

Execute Contingencies

The AST program may be solid and execute perfectly, but there are always more conditions that can be tested than time allows. In the design phase, there should have been a prioritized list of requirements. Once the core requirements have been covered in test cases, it may be helpful to look into contingencies from the baseline test case.

Example contingencies are scaling for many more users, applying tests in different environments, and using different hardware and operating systems. For example, an automated test (GUI-based) ran fine in the SIL; but, when attempted at another location it failed because the laptop used was older with lower screen resolution. Another important contingency is when the SUT is running in a degraded state. These contingencies may necessitate changes to the AST framework and scripts. The benefit is that the enhancements make the automation more robust and often will reduce the maintenance burden.

Phase 4: Analyze

The analysis phase takes the output from the executed test cases and transforms it into decision quality information to assess both SUT software performance and the AST framework effectiveness. The team of testers, software developers, software testers, and automators may require the assistance of an analyst familiar with scientific test and analysis techniques (STAT) and other applied statistics methods.

Establish Data Output Format

A quality AST program requires a detailed account of everything that went on during the execution. This serves multiple purposes to include quick fault isolation, improved debugging, comprehensive understanding of state transitions, and confidence the AST framework is executing properly. The most important attributes are transparency into the system status/execution steps and traceability back to requirements.

Representative output products include detailed logs (possibly color coded to indicate faults), screenshots of system status upon failure, and data written to output files. Critical items to be included

in an output file include:

- Test case ID
- Requirement ID
- Test data steps and current state
- Expected results
- Actual results
- Pass/Fail

Bottom line: Taking time up front to establish output files will set your automation up for success. Choice of tools may dictate what type of products are possible.

Analyze SUT Anomalies

The purpose of software testing is defect discovery. Software bugs, faults, errors, and anomalies will occur – it is important to not only detect the error but also to determine the source of the discrepancy. With a properly designed AST architecture, the output reports should allow rapid fault isolation. If this is not the case, then a formal root cause analysis program should be instantiated. Some techniques are to ask why 5 times (the 5 Whys), cause-and-effect (Ishikawa) diagrams, and the 8-step process from the auto industry.

Determining whether the anomaly is attributable to the SUT or the AST framework is critical. For example, an automated test stopped because the image of the user, "Gregg" was not correctly identified; however, it was correctly entered from the input file (Gregg, 2014). Other concerns are the overall error rate. False positives occur when the system detects an error that does not exist. Similarly, we want to minimize the occurrence of false negatives where software defects go unnoticed. On a related issue, we do not want to react to every low-level issue that does not have an impact on performance. For all valid defects, software trouble reports should be generated and socialized in a collaborative environment so all team members have visibility into the issues.

As part of post-processing, use the software defect information gathered to improve similar or identical future testing (e.g. regression), or to better scope next acquisition phase of testing (e.g. DT to inform IT Verify Requirements Adequately Tested).

Some useful references for root cause analysis are:
www.au.af.mil/au/awc/awcgate/nasa/rootcauseppt.pdf and https://nsc.nasa.gov/SFCS.

Bottom line: Set your team up for success to easily identify anomalies, balance false positive and false negative reporting, and produce solid documentation to pass along to subsequent testing.

Summarize Requirements Tested and Identify Possible Voids

The test log should be robust enough to determine what requirements actually were tested. It should also provide the levels of the input variables that would allow the team to quantify how well the test region was covered. Review the system requirements and determine gaps or areas still needing to be

tested. The team should provide initial data analysis findings and conclusions for reporting on the requirements achievement. Be sure to include discussions on the coverage of the test space and stress the value of automation to the overall test effort.

Check Repeatability and Reproducibility

It is common in practice that the same tool and same test do not provide the same results all the time. This is especially true for GUI testing. The team will need to test the stability of scripts to see if they give the same output each time. Repeatability refers to the same tester getting consistent results upon executing the same routine. Reproducibility accounts for variation in equipment. For AST, repeatability would be getting similar results from the same tool or suite of tools and reproducibility would be consistent results across multiple tools and solutions. Negative testing triggers failures to determine if the software responds correctly.

Bottom line: Automation can be frustrating because many solutions are brittle and do not provide the expected results from the same tests. Search for robust tools and methods that have demonstrated success on similar systems. Beware of Capture-Replay not working across different platforms and operating systems.

Compute and Update Automation Metrics

The estimates of automated test benefits required in Phase 0 were helpful to inform a 'Go/No Go' decision. Now that automation has occurred, the team needs an accurate, reliable, and realistic measure of how effective the effort was for the affected test program. This metric will inform the updated ROIs and the direction of future AST work. Some metrics are:

- Percent requirements tested
- Increased coverage in lines of code tested
- Increased coverage of expected operational paths and use cases
- Time to run the automated test sequence
- Time to develop automation capability; time to execute manually
- Defect discovery rate
- Testing time on off hours
- Reusability of automated scripts

Bottom line: Leadership is always interested in the business case for automation as they look to expand the program or discontinue. Continually update metrics that capture how well automation is working relative to the manual tests.

Compute ROIs and Consider Future AST Program

After the initial investment to achieve AST capability, an evaluation needs to be made on its worth. The team has a variety of options. Some of the more prominent ones to consider are listed below:

- Go deeper with the current effort
- Scale up with more user load
- Automate new requirements
- Adopt different tools
- Transition from GUI to code-based API tests
- Conduct contingency test cases
- Abandon automation

This decision puts the team back to Phase 0: Pre-Planning, but they are now much more informed. They should review the research and decision-making accomplished during the pre-planning phase given the valuable AST journey they completed. Use the metrics to perform cost-benefit analysis of automation versus manual testing and report the findings with recommendations.

Phase 5: Maintain

The STAT test methodology does not include a Maintain phase. Unfortunately, empirical evidence suggests that once an automated test is executed and analyzed, the job is not done. A pervasive theme across all services is how much effort is required, though not necessarily resourced, in the maintenance phase. Changing SUT configurations, updated tools, new tools, and adding new personnel are just a few of the dynamics that require an updated AST solution.

Manage Automated Software Suite Configuration

Configuration control is essential. Systems and architectures are not stable for long across the DoD enterprise. Fortunately, the design phase already has prepared the team by having them design a configuration control system to track changes over time linked to test cases and requirements. There are many file management system options and a disciplined approach is needed.

It is rare that a change to the SUT would not require some change to the automation. Think of a GUI-based AST – it only takes a minor change in the expected output graphic for the automation to report an error of not finding the image. A key question is "*how to best manage updating the test script over time.*" Different SUT configurations will correspond to specific AST framework versions. Another question is whether a single individual should be responsible for developing and executing the scripts or whether an automation team collectively updates the scripts. Periodic verification/validation programs are necessary to ensure compatibility between the SUT and automation solution.

Update Automation Code

As previously mentioned, the automation code needs to be responsive to changing SUT and interfaces. The team needs to be aware of any changes that would impact execution and should expect they will not be aware of many of these changes until they are executing a test. Code should also be updated to account for new tools and updates to existing tools. Code may need to be modified to incorporate better coverage or improve other metrics.

The design of the code and scripts should be nimble enough to easily account for frequent changes. This is not the case in practice, but many automators and tester wish they would have developed the solutions with a "design for maintainability" mentality to meet changing system output or to better improve automation metrics.

Manage Scripts

Each program should create their own script repository. They will need to access scripts to update them and should track the different versions under the configuration control system. The team can share scripts internally with other developers/automators or externally with other organizations who may be able to reuse them for efficient automation. The script repository should also include acquired scripts from research efforts and other organizations. These can be catalogued for ease of adaptability to new testing needs.

Track Program Software Defects

A disciplined approach to tracking software defects is necessary to fully realize the benefits of software testing. A closed-loop system such as a Failure Reporting, Analysis, and Corrective Action System (FRACAS) database provides excellent visibility and accountability. Some fields include conditions such as failure occurred, time/date, system status, initial corrective action, point of contact to own failure, and current status. Many of the defects will be subject to Failure Review Board (FRB) actions, and the FRACAS database is set up to provide required background information and record FRB recommendations. The team should institute a process as part of the regular battle rhythm to update and review the FRACAS database.

Assess Defect Discovery Trends

It is helpful to classify the software failures based on mission criticality. This serves two purposes: to quantify reliability in terms of defects per thousand lines of code and to prioritize failure modes. The FRACAS database should allow the team to view defects over time and to assess how reliability is improving or degrading. Data visualization methods can easily stratify based on failure mode, failure criticality, requirement, test case, version, and so forth to see if there are trends over time. In addition, there are statistical models that can help quantify system performance.

If the SUT program management has placed value on finding and correcting software failures, then the defect rate should decrease. There are several software reliability growth models outlined in IEEE 1633, Recommended Practice on Software Reliability (2016, Annex C Supporting Information on Reliability Growth Models) that can be used to predict future failure rates based on the current failure rate, management aggressiveness in finding root causes, remaining test time, and the quality of the contractor's software development program as measured by capability maturity model integration (CMMI) rating.

Bottom line: Plan for future automation success by being aware of 'level of effort' required to maintain your automation capability. Realizing that the true power of automation lies in repeated application of the same or similar testing – it is imperative that minimal modifications are required to your library of

functioning scripts. Updates to scripts should be minor for either of the following purposes: a) to execute the same test later when the SUT or test environment changes, or b) to adapt scripts to perform similar but different automation purposes. Know that maintenance loads can vary significantly depending on the automation tools selected.

References

Aldor-Noiman, Sivan, Paul D. Feigin, and Avishai Mandelbaum. "Workload forecasting for a call center: methodology and a case study." *Annals of Applied Statistics*, no. 4 (2009): 1403–1447.

Ammann, P. and Offutt, J. (2017). *Introduction to Software Testing, 2nd ed*., Cambridge, New York, NY.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, *41*(5), 507-525.

Bartholomew, R. (2013). An industry proof-of-concept demonstration of automated combinatorial test. In *Automation of Software Test (AST), 2013 8th International Workshop on* (pp. 118-124). IEEE.

Binder R., Carney, D. and Novak, W. (2015). *Navy Software Test Automation Study: Product and Technology Background*. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2015-SR-047.

du Bousquet, L., Ledru, Y., Maury, O., Oriat, C. and Lanet, J.L. (2004). A case study in JML-based software validation. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on* (pp. 294-297). IEEE.

Dustin, E., Garrett, T., Gauf, B. (2009). *Implementing Automated Software Testing: How to Save Time and Lower Costs*, Pearson Education, MA.

Fewster, M. and Graham, D. (1999). *Software Test Automation*, ACM Press, Edinburgh, UK.

Graham, D. and Fewster, M. (2012). *Experience of Software Test Automation: Case Studies of Software Test Automation*, Pearson Education, NJ.

Gregg, Brendan (2014). *Systems Performance: Enterprise and the Cloud.* Pearson Education, NY.

Kruse, P. (2016). Test oracles and test script generation in combinatorial testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on* (pp. 75-82). IEEE.

Kuhn, D., Hu, V., Ferraiolo, D., Kacker, R. and Lei, Y. (2016). Pseudo-exhaustive testing of attribute based access control rules. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on* (pp. 51-58). IEEE.

Kuhn, D., Kacker, R., Lei, Y. and Torres-Jimenez, J. (2015). Equivalence class verification and oracle-free testing using two-layer covering arrays. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (pp. 1-4). IEEE.

Kuhn, D. R., & Okun, V. (2006). Pseudo-exhaustive testing for software. In *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA* (pp. 153-158). IEEE.

Liu, H., Kuo, F., Towey, D. and Chen, T. (2014). How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, *40*(1), pp.4-22.

Mathur, A. P. (2008). *Foundations of Software Testing, 2/e*. Pearson, NY.

Microsoft Corporation (2007). *Performance Testing Guidance for Web Applications,* Microsoft, WA.

Molyneaux, Ian (2015). *The Art of Application Performance Testing: From Strategy to Tools, 2/e.* O. Reilly Media, NY.

Mosley, D. and Posey, B. (2002). *Just Enough Software Test*, Prentice Hall, NJ.

Page, A., Johnston, K., and Rollison, B. (2009). *How We Test Software at Microsoft*, Microsoft, WA.

Paskal, G. (2015). *Test Automation in the Real World: Practical Lessons for Automating Testing*, MissionWares, CA.

Pedron, L. (2015). *Software Test Automation: Getting Started Guide for QA Managers, Quality Engineers, and Project Managers*, Independent Publisher.

Appendix A: Acronym List

Acronym	Description	Acronym	Description
AFB	Air Force Base	**JWICS**	Joint Worldwide Intelligence Communications System
API	Application Programmer Interface	**LVC**	Live, Virtual, Constructive
AST	Automated Software Testing	**NMCI**	Navy Marine Corps Internet
ATRT	Automated Test and ReTest	**OSD**	Office of the Secretary of Defense
CAC	Common Access Card	**OT**	Operational Test
CDRLs	Contract Data Requirement Lists	**POA&M**	Plan of Action and Milestones
CMMI	Capability Maturity Model Integration	**POC**	Point of Contact
COE	Center of Excellence	**RITE**	Rapid Integration and Test Environment
CRON	Command Run On	**RM**	Requirements Management
DASD(DT&E)	Deputy Assistant Secretary of Defense, Developmental Test and Evaluation	**ROI**	Return on Investment
DEF	Developmental Evaluation Framework	**ROM**	Rough Order of Magnitude
DEO	Developmental Evaluation Objectives	**SIL**	Software Integration Lab
DI2E	Defense Intelligence Information Enterprise	**SIPRNet**	Secured Internet Protocol Router Network
DoD	Department of Defense	**SME**	Subject Matter Expert
DT	Developmental Testing	**SPAWAR**	Space and Naval Warfare Systems Command
DT/OT	Developmental Testing/Operational Testing	**STAT**	Scientific Test and Analysis Techniques
FAA	Federal Aviation Administration	**SUT**	System(s) Under Test
FRACAS	Failure Reporting, Analysis, and Corrective Action System	**T&E**	Test & Evaluation
FRB	Failure Review Board	**TDD**	Test-Driven Development
GUI	Graphical User Interface	**TEMP**	Test and Evaluation Master Plan
HP	Hewlett Packard	**UFT**	Unified Functional Testing
IDT	Innovative Defense Technologies	**UML**	Unified Modeling Language
JML	Java Modeling Language	**VV&A**	Verification, Validation, and Accreditation