ETALIQ
intelligent quality

Send, Receive, and Verify!

Part I: Why Isn't Automation That Simple

A White Paper
by Etaliq Inc.

www.etaliq.com

Etaliq Inc.
4B–2548 Sheffield Road
Ottawa, ON K1B 3V7

Phone: (613) 241–1385
Fax: (613) 241–1523
http://www.etaliq.com

Synopsis

Send, Receive, and Verify! The perception is that automation is that easy, or at least it should be. Telecommunications manufacturers and providers use automation to verify the operation of their products and systems prior to customer release or production implementation. This document describes the predominant system of choice for telecommunications automation systems and many of the productivity challenges included in its selection, implementation, and architecture.

The vast majority of these telecommunications automation systems are built in-house. The preferred tools are Tcl/Expect, Perl, and regular expression scripting languages which are most often implemented on a Unix workstation or server with additional shell scripting environment setup utilities. Source code version control and change management is most often accomplished with CVS or RCS. These scripting languages and utilities, which are all 1980's vintage tools, have undergone very few of the productivity enhancements that the product developer IDE tools have seen in recent years. They are merely the starting point of what can only be described as a patchwork system of languages, utilities, tools, environment settings, and libraries, cobbled together over a number of years.

There have been a rash of acquisitions in recent years with large technology and tools companies acquiring test automation tool suppliers such as Rational (IBM), Mercury (HP), and Segue (Borland), to name a few. To date none have been able to provide a product that meets the requirements for a telecommunications automation system.

Over the last few years, several new companies have arisen attempting to solve some of these challenges; many of them by putting GUI front ends on these existing utilities, as well as adding a few productivity functions to reduce the time required to build and maintain simple scripts. Upon review of the architectures and script building capabilities of these newer tools, many of the productivity challenges remain un-addressed, while using them in scaled stress and performance testing (more than 100's or 1,000's of interfaces and sessions across a series of nodes) remains unattainable.

It is clear that what is needed is a new approach: an architected solution that no longer requires that Automation Engineers be used in tandem with Subject Matter Expert telecom Test Engineers to implement and maintain automated tests and scripts. A new solution should assist in ensuring that the test code matches the descriptive Test Plan objectives and procedures. Complex multi-device communication, simplified "Send, Receive, and Verify!" functions, as well as procedures for testing with 1,000's of connections and interfaces must be integrated and easy to use. It must more efficiently use lab resources and enhance the log review and failure cause analysis processes, both during test creation and regression execution. This new tool must also include integrated management reporting functions to summarize results and resource usage. It should no longer be necessary to have highly qualified engineers entering zeros and ones in spreadsheets. Summarized results must be integrated and customizable for management reporting. For example, reporting globally from run to run, day to day, week to week, providing a view of the progress of a project. These same results must be filterable by project, sub-project, software version, system component or category, category of test, and category of hardware and/or technology.

Simplify...
Simplify...
Simplify...
Send, Receive, and Verify!
It should be that easy.

This document describes in significant detail, with examples, an overview of the testing cycle while highlighting the inefficiencies, trials, and tribulations of the current telecom automation systems. The specific examples are drawn from the experiences of the writers as witnessed in both manufacturing and provider production and lab facilities.

Watch for part II of this document titled "Automation is That Easy with ETA." It describes solutions to most of the challenges identified herein as well as a series of productivity enhancements you won't want to miss.

Challenges

Test Cycle Overview

The testing cycle begins with a review, by a senior Subject Matter Expert (SME) Verification Engineer, of the customer and system requirements documents. This engineer in turn creates a Project Testing Charter/Architecture document which provides an overview of all testing requirements for the new product or system. This document includes specifications for lab setups, an overview of Test Plans to be created, performance and stress test requirements, a list of regression tests and scripts to be executed during the project to ensure unaffected sub-systems are indeed unaffected, and an initial estimate of required testing resources and timelines. This Test Architecture document is then used by other Project SME's to build lab setups and write detailed Test Plans and Cases. Of these Test Cases, 20 to 40 percent are then selected for automation.

Test Plan documents are reviewed by developers, testers, and other project participants in accordance with ISO processes. Test labs are prepared in advance, so that upon receipt of early product code, manual feature testing begins. Automation of the selected Test Cases begins shortly after the product gains a reasonable level of stability, as verified by manual testing. Problems are documented and Test Case results are tallied in order to represent the progress of testing and the quality of the product to management. When feature testing completes and quality objectives are in sight, the product proceeds through stress, performance, and regression testing, then to early field trial, and finally to general customer release. All documents, results, logs, and summarizations are filed in compliance with ISO standards. Automated scripts are made production ready for regression purposes and a project post mortem follows.

Post mortem analysis documents identify and describe the project successes and failures, which most often result in modifications to be used for subsequent projects.

Test Plan/Case Documented Requirements Vs Test Script Implementation

Often, an audit of a selection of the new automated scripts versus the Test Plan requirements is undertaken during the post mortem phase of the project. This audit periodically includes a review of some of existing regression scripts, used during the project. These scripts are also reviewed from a script content vs Test Case requirements perspective.

Results of these audits have been astounding. Post mortem documentation has shown that up to 45% of the Automation Engineer (AE) written new scripts did not meet the objectives defined within the SME written Test Plans and Cases. Results for regression scripts were even more disturbing: where some of these, as well, no longer met the documented Test Case objectives. Here are some examples.

Scripts that had undergone changes during normal execution and maintenance, no longer met the original SME written objectives for the Test Case. Root cause analysis most often showed that Automation Engineering and regression test maintenance staff did, in fact, to the best of their ability, implement procedures that performed the steps and objectives defined by the SME. The cause of the mismatch was traced down to a lack of expertise on the part of the automation and regression staff when interpreting and converting SME written steps into automation code.

Other cases showed that modifications to existing library procedures to meet the requirements of new tests, had negatively affected the operation of other production regression scripts. These backward compatibility issues only came to light during subsequent production regression runs resulting in the waste of valuable lab execution and regression engineer research time.

Still, in others, some of the SME written procedures within Test Cases, were slightly modified in order to reduce automation effort and keep on schedule. These changes were deemed by senior automation staff to have little or no effect on the script meeting original test objectives. Again this mismatch was traced to a lack of expertise on the part of the automation staff where their lack of in-depth product and feature knowledge contributed to the error.

Occasionally, existing library procedures were selected and used within the new Test Cases based on their library contained descriptions. While the description seemed to match the written requirement in the Test Case, no additional research was done to ensure an exact match. This particular mismatch was traced to out-dated function descriptions contained in production libraries.

All of these scenarios apply to both new and regression scripts to varying degrees. In the case of regression scripts, the older the script was the less likely it was to have accurately met the objectives in the original Test Case.

There appears to be a fundamental flaw in the operation of test and automation organizations. In nearly all cases, SME Test Engineers and Automation Engineers are not the same individual. Test scripts are seldom, if ever, audited by SME Test Engineers. Scripts rarely remain in step with documented test requirements. The written Test Plans and Cases are reviewed by the Development Engineers (DE) who ensures that the SME's have properly defined tests that completely verify the correct operation of the product. Yet, the scripts, library procedures, and test log results are not reviewed by any of the Test or Development Engineers, thereby creating a possibility of mismatch.

During Test Automation audits, very often the code *no longer matches* the Test Plan objectives that it was designed to test.

This can result in *false passes*, and a false sense of security.

Scripting environments today are implemented as a series of libraries of procedures that are modified, as required, during new feature and regression test cycles, for maintenance purposes. These modifications proceed without review of, or regard for, the original SME written Test Cases. Sloppy and/or misguided modification to the procedures and descriptions in these libraries cause many failures in both regression and project automation runs.

SME written Test Plans and Cases are infrequently designed with automation in mind. For example, a SME will write a step in a procedure stating, "Verify that all affected L2VPN connections return to an up state, with statistics counters increasing verifying that traffic flow resumes." Automation environments are written such that this simple step in the Test Case requires as many as seven or eight sub-steps of automation code to complete. These sub-steps will be repeated within a loop for all affected L2VPN connections in the Test Case, and followed by traffic analyzer statistics verification. The sub-steps required to accomplish this are as follows: 1) Loop on a list of all affected L2VPN connections; 2) Perform the required show commands for state and statistics verification; 3) Parse the output of each show command and retain the values for verification; 4) Verify the state portions of the L2VPN connection in these show commands ensuring that all of the appropriate local and remote state values therein, show them to be "up"; 5) Repeat the show command for statistics; 6) Parse these statistics and retain them in a second series values for verification; 7) Loop on all of the appropriate statistics that verify that the values are either increasing as required, or not increasing as appropriate; 8) Upon completion of all L2VPN connections, retrieve and verify the traffic generator analyzer statistics.

Upon review of the type and amount of code necessary to accomplish a relatively simple step in a Test Case, the solution becomes obvious:

Simplify...
Simplify...
Simplify...
Send, Receive, and Verify!
It should be that easy.

It is obvious that a plethora of challenges exist in writing and maintaining scripted Test Cases that continue to meet the objectives defined by the SME Test Engineer. Reducing complications in script coding would contribute heavily to greater efficiency and reduced maintenance. Simplified "Send, Receive, and Verify!" functions are required. Simplifying this code creation process would allow many more scripts to be created earlier in the test cycle. If the new test code

could be tested and verified prior to product availability, automation would be available early in the test cycle rather than at the end. Also, more lab resources would be available for test executions during the test cycle.

Test Cycle and Automation: Lab Resources

Today, test automation environments and tools require that the product to be tested be available, in a near stable form, in order to verify the operation of the test scripts. With time-to-market being a primary key to success in the telecom market, opposed by continued pressure to reduce costs (both human and lab), there is limited flexibility during a test cycle. Resources to repeatedly test a significant portion of the Test Cases, either manual or automated, are unavailable. Resources to automate more Test Cases are also unavailable.

This limited flexibility contributes greatly to even more challenges and inefficiencies in today's overall test productivity. A test cycle often includes hundreds or even a thousand Test Cases for execution. Of these, only a small portion (20–40%), are selected for automation. Valuable lab resources are used to create and verify automated Test Cases during the test cycle rather than having a set of pre-built, pre-verified automated cases ready at the start. In addition, even if more Test Cases were automated, more regression resources would be required to manage the job and schedule execution parameters, and more lab resources would need to be available. The reality is that there are not enough lab resources available to repeatedly run all of the automated Test Cases, either at the beginning or end of a test cycle, or for regression purposes.

The requirement to have the product available to verify automation means that new and modified test scripts are built and verified only during a test cycle. It is due to this fact that automation is predominantly used at the end of a test cycle and onward for regression purposes rather than at the beginning, where continued use could contribute to more rapid achievement of quality objectives and reduction in overall test cycle duration.

It is highly desirable to create *more automated Test Cases*, if they can be used efficiently during a test cycle and afterwards.

During a test cycle, lack of new available automation and human resource constraint means that only Test Cases affected by bug fixes are repeated. This, combined with the fact that not all Test Cases are automated—and even if they were automated there is not enough time to run them—means that many of the Test Cases are never repeated, *ever*!

> **It is highly desirable to *execute validated automation much earlier* in the test cycle. If only it were possible!**

It is obvious that a simplified coding strategy and some form of ability to validate scripts prior to product availability would result in usable automation much earlier in the test cycle. A side affect would be that additional hardware resources would be freed up for testing rather than reserved for script creation and verification. Additionally, more efficient management of execution scheduling profiles would enable more tests to be run more frequently.

> **If test automation code could be *built and validated* prior to the availability of *newly developed product*, then: 1) Valuable lab resources could be freed up to perform *more tests* during the test cycle, rather than only at the end; 2) More automation could be available to *test more frequently*.**

Today's Automation Infrastructure Inefficiencies

Automation infrastructures implemented with 1980's vintage scripting tools, have seen few to none of the enhancements that products used for product development or operational monitoring have experienced over the last 30 years. Tools today for development and operations are a far cry from those used in the 1980's, yet test automation infrastructures in the engineering disciplines for Telecommunications, Defense, Aerospace, and Medical remain essentially unchanged.

Inefficiencies, in these test automation infrastructures, have far reaching effects in all aspects of product engineering development. This includes time-to-market, product quality, human and hardware resource costs, not to mention customer found defects and corporate reputation.

These infrastructures are implemented with a mishmash of the following: Unix shell script environment variable files and settings, execution or job file parameter files, lab device related communication and hardware setup information files, Tcl/Perl scripting environment initialization libraries and functions, communications library functions for Expect/Perl, a reporting library with a series of functions for creating log and report files, a multitude of specific function libraries and procedures for configuring/parsing/verifying various sub-systems within the product being tested, and one or more libraries for testing device specific functions including traffic generators/analyzers.

In many of the medium to large test automation environments, various files/libraries/procedures and functions were created in the mid to late 1980's and have continued to grow unabated. In a couple of cases, where the test automation infrastructures were being audited, more than 250K lines of code were being loaded to run an individual script with an initialization time of more than 7 minutes. That is to say, that prior to running the first script related line of code, 7 minutes of wasted lab time was consumed. The bloat within these systems has a dramatic affect on lab resource usage, not to mention the amount of time spent by an Engineer when developing, debugging or running a script for regression purposes.

Systems have grown to well beyond a bloated state. System initialization and per-command *execution times* must be reduced, resulting in *more efficient* use of all resources, lab and human.

Today's test automation infrastructures, implemented with 1980's vintage scripting tools, lack many of the efficiencies that would enable them to run more quickly and achieve more test results in a shorter period of time. As an example, one of the most prevalent inefficiencies

in today's infrastructures is that they do not provide a quick and easy to use "Send, Receive, and Verify!" capability. That is, existing infrastructures must implement a seven or eight sub-step process to achieve a simple test step within a Test Case. See the example above showing the sub-steps necessary to "verify that all affected L2VPN connections return to an up state, with statistics counters increasing verifying that traffic flow resumes." This means that the time required to develop a correctly operating script is much longer and more complex than it needs to be.

Simplify...
Simplify...
Simplify...
Send, Receive, and Verify!

Today's automation infrastructures seldom include the ability to verify script integrity through a simple compilation and syntax check functionality. The 1980's vintage scripting tools require that Automation Engineers use valuable lab resources to actually run the script to identify even the simplest of coding errors. Coding errors as simple as undefined variables or misaligned quotes are only discovered during execution. These scripting tools are "run to completion" batch oriented programming systems where, upon discovery of individual errors, they abort without assisting in identifying even two errors at a time. To identify each individual error, the Automation Engineer must run the script, repeatedly, until all coding errors are found and corrected. After correcting all of the syntax-related errors, they must continue running it to identify and remove all of the, more complex, logic errors. When feature or product changes require it, enhancements or fixes to existing library functions and procedures must be done with such care as to not affect other scripts and procedures that use them. The problem is that the library procedures often call other library procedures, making it near impossible to know which scripts are affected by a library change. Again, all of these contribute heavily to an inefficient use of valuable resources, both human and lab hardware, during a test cycle and beyond.

Systems should include *pre-execution compilation and syntax verification* in order to allow *more efficient* use of all resources, both lab and human.

Another of the severely unproductive components of automation infrastructures today is related to the logs and reports created during execution. Log files created during execution include; a programmed log output file where script progress and errors are noted, console log files for each lab device, summary logs for Pass/Fail results and execution information, and a more detailed log of test progress where information about test steps, library procedures, debugging and select device output are logged. As many as 10 to 20 individual files are created for each execution. The problems here are; none of these files are linked to each other, output to each is generally free form text, little structure or built-in ease-of-use information is included, errors, warnings and problems are in no way linked to the source script or library function executed to generate them, and, finally, each Automation Engineer often has total autonomy in deciding what and when information gets logged to each file. Regression operations staff requires several days or weeks of familiarization time with the log output and script coding, prior to being assigned production operator for that script. Test executions that run 20 tests create logs that are often 10 MB or more of raw text data, spread over a set of 10 to 20 files. Without the files linked to each other and to the origin source code, finding an error 3 MB into the detailed log file, for example, causes great difficulty in determining which test within the script was being executed at the time of error. If the error seems to be linked to a function call, it is difficult to know which function, within which library, was called from within which test, for what reason. Log files, linked to each other and to the source files would make it infinitely more productive to do problem determination. All this to say that, the log review mechanisms, built into existing automation infrastructures, leaves much to be desired.

System log files should be *integrated and structured* to more *quickly identify failure points*. Unstructured output files, that are not linked to each other nor to the origin source code, are an *inefficient use of automation resources*.

Why can't Automation be as simple as *Send, Receive, and Verify!*

System Architecture: Test Automation Supporting Tools

Additional tools are required to efficiently run a fully architected test automation solution. At a minimum the system must include an execution scheduling and prioritization mechanism, a summarized results reporting database, node usage tracking and reservation features, log output review, compression, storage and retrieval functions, and a source tracking mechanism for Test Plans, Test Cases, execution parameter files, node definitions and inventory information.

These complimentary tools are also created and maintained in-house, using non-integrated applications, built with yet another set of programming tools (often Java, or HTML), each with its' own non-integrated back-end database, with even more tools being added on an as-needed basis. The ideal scenario would be a fully integrated and thoughtfully architected Test Automation Infrastructure that includes all of these complimentary tools. The reality is, however, that integrating these in-house developed systems is not a priority.

In the last couple of years, some users have begun migrating to commercial off-the-shelf products that do specific parts of an automated solution. Use of these tools is catching on to reduce in-house tool maintenance and development costs.

> ***A fully integrated and architected test automation system*, including Dispatcher and Execution Engine, Resource Reservation and Scheduling, efficient Execution Log Reporting, flexible File Management, customizable Summarized Pass/Fail and Node Usage Reporting, with interfaces to various other test and inventory management tools, *is required*.**

Conclusion

A truly innovative test automation infrastructure solution would provide the ability for SME Test Engineers to automate themselves, without the need to team up with Automation Engineers. This would eliminate miscommunication between the Test and Automation Engineers, resulting in less scripts that do not meet test objectives and many less *false pass* automated Test Cases. Merging the Test Plans with the automation code using this easy to use infrastructure would dramatically reduce mismatches and false passes. A simple to use system based on the principle to "Send, Receive, and Verify!" would contribute to increased accuracy, reduced time to create and execute, and ultimately to a highly efficient automation operation. Development engineers could review the automation at the same time as they review the Test Plan, thereby extending ISO processes into the automation realm, again contributing to increased accuracy.

An ability to verify Test Code validity prior to product availability would dramatically increase the usefulness of automation. The test code would be available much earlier in the test cycle, dramatically reducing the amount of lab resources used to verify automation code during a critical phase of every project.

An integrated compiler and syntax checker would eliminate much of the problem determination time during new test development and existing test maintenance, by not wasting time using valuable lab resources. In addition, a truly innovative test automation solution could provide the ability to verify the syntax and the operation of newly developed automated tests, long before the product to be tested is delivered.

A fully integrated flexible smart scheduler would optimize the use of lab resources and make it possible to run more tests, more frequently. The node usage portion of this scheduling system would document resource usage and assist in identifying and rectifying inefficiencies.

Full integration of all the environment setup, device definition and inventory, tool setup and initialization, and all of the various library and procedure files would dramatically reduce the time to create, maintain, and run automated tests. This, in combination with an

optimized automation infrastructure, would mean that more tests could be run, more frequently.

Structured logs and reports, that are linked to each other as well as to the origin source files, would dramatically reduce resource efforts during problem determination; making all resources more productive. This also provides a side benefit where customer-reported failures can be traced to actual log and source files that should have detected the failure.

An integrated, customizable, summary reporting mechanism would mean that more detailed and accurate statistics about product, feature, or individual project quality are available at the touch of a button.

To date, tools like IBM Rational, HP Mercury, Borland Segue, and others, have been unable to break into this market where complexity rules. At last count, more than 100 companies claim to provide, meet or exceed some or all of the requirements to operate an efficient automation infrastructure for this environment. Additional evidence can be found in the number of patent submissions related to test automation being received by both International and U.S. Patent Offices. This market still searches for a viable solution even after 30 years of evolution in the development, test and operation tools field.

When looking to a new automation infrastructure tool, narrowing the field of viable alternatives is fairly simple. Ask them for an example performance, stress and scalability test. Define for them a fairly complex lab setup with 4 or more nodes, each having multiple operator sessions, and all physically interfaced to each other. Request an example Test Case with code to verify 1,000 interfaces of varying classifications, with a homogeneous L2VPN configuration and static routing. Include a single traffic generation stream for statistics verification, and "verify that all affected L2VPN connections return to an up state, with statistics counters increasing verifying that traffic flow resumes." This will reveal the time to create a complex test, the time to run such a test, and the lab resource usage required. This will eliminate all but the serious players. Of those that remain, ask that the test be run without connecting to the lab devices, as though they were building a set of tests for a product that is not yet available for testing.

Any vendor of test automation infrastructure products must provide a tool that addresses all of these problems. Additional productivity features should also be present and include the ability to integrate existing scripting facilities as well as provide value-added functions for development, execution, log review and reporting. To simply put a pretty GUI front-end on an existing Tcl/Expect/Perl automation infrastructure and provide the ability to automate simple tests or tasks is *not* enough.

Test organizations that address all of these issues will attain the following results: increased test resource efficiency, reduced time-to-market, higher quality targets achieved faster and earlier, reduced customer-found defects and, ultimately, positive contributions to corporate reputation.

For a solution to all of these identified issues and many more, visit `http://www.etaliq.com`.