An Open Framework for Managed Regression Testing

Naina Mittal and Ira Acharya

Tata Consultancy Services, D-4, Sector 3, Noida – 201301, India
{nainam,iraa}@delhi.tcs.co.in

Abstract. In the prevailing competitive environment, companies are facing tremendous market pressures to launch defect-free products in a timely manner. This challenge is compounded when a product runs into sustenance phase because complete regression runs need to be performed for each change/enhancement made in every build/release of the product-under-test. This paper discusses an architectural framework approach to address the above challenge, thereby aiming to make regression testing a simple, repeatable and automated exercise. The framework design encapsulates hierarchical test case management, multi-user support, product version maintenance, and automated test execution and result analysis to facilitate easy testing. The open architecture of the framework allows it to augment capabilities of some other testing tools by providing adapters to them, thus, eliminating the rigidity of use of a particular tool. The paper also draws a comparison of the architectural approach with other existing frameworks and presents a cost-benefit analysis of the suggested approach.

Keywords. Test automation, managed testing, test planning, test execution, test framework, networking equipment, regression testing, black-box testing, test scripts, test bench, hierarchical test case management, test plan tree, framework deployment, test-cycle reduction, testing tool collaboration.

1 Background

In the new economy, testers face the challenge of ensuring timely, defect-free product launches in the market. In this world of ever shortening internet times, minimal time-to-market is the key for steering ahead of competition. The goal of achieving this difficult target puts a direct liability on the development and test cycle times of products-under-test. This is almost impossible without defining a streamlined process for testing and incorporating desirable automation levels at every stage. A common problem faced by most test managers is that of managing the test process itself, which is a unique challenge, requiring judgment, agility and organization. In spite of sizable investment in tools (for example, traffic generators, voice simulators, script generators) that aid in testing of the product-under-test, test managers still face the dilemma of managing testing using these disparate entities to achieve an optimal testing solution. Test management challenges include establishing a communication mechanism between these test tools, controlling test planning, test execution and test results from a central control point, tracking test cycles of multiple builds/releases of the product-under-test, arranging tests in an organized manner so as to optimize on time as well as

D. Hogrefe and A. Wiles (Eds.): TestCom 2003, LNCS 2644, pp. 265–278, 2003.

hardware resources, automating execution of tests and much more. [1], [2]. Thus, while the investment in testing tools may appear as the apparent solution to the problem, it is an even bigger problem putting them to use effectively and preventing them from becoming shelfware.

2 Challenges Faced in a Manual Testing Scenario

A manual test system is a costly exercise in terms of time and effort. It is usually characterized by unmanaged testing, inconsistent result logging and inefficient defect-management. Moreover, there is usually no provision for tracking test cycles of various product versions which maybe undergoing testing simultaneously. In spite of dedicating a good-sized testing team for product testing, there is always the likelihood of defects left unnoticed in the shipped deliverable, which are later discovered and reported by the recipients of the deliverable. This situation is unwelcome for every product vendor, as defect detection after delivery not only has a direct impact on cost, it also implies vendor's inability to contain defects.

The need for a test management framework stems from the aforementioned problems that are associated with a manual-testing scenario. In the context of telecom/networking equipment, testing is an even more complex proposition because it involves setting up test benches and having a number of third-party test equipment in each test bench required to aid testing of different functions of the product-under-test and to simulate multiple test scenarios. This pre-test setup is followed by manually configuring the product-under-test as well as peripheral test equipment before running each test case. Finally, a test case is executed and the results as observed on the console are logged for reference. The complex nature of these tests makes their execution susceptible to errors arising due to either incorrect configuration of peripherals or incorrect test execution by testers who are likely to commit errors either due to inadequate domain knowledge or simple negligence. A mid-size system can have large number of possible test scenarios, which when tested manually take significant amount of time.

Thus, having a managed test framework to control the test management tasks, integrate the various testing tools and automate test execution resulting in better quality deliverables would make the test management process simple.

3 Addressing the Most Cumbersome Stage of Testing

Products go through multiple stages of testing in their lifetime from conception to release. Producing reliable systems requires a well-planned, comprehensive application of several techniques throughout the development life cycle, collectively known as Verification and Validation [3]. The basic stages of testing include:

- Unit Testing
- Module Testing
- Sub-system Integration Testing
- System Testing
- Regression Testing

Although it is desirable to have maximum possible automation at each stage of testing, it is also important to consider cost for introducing automation at every stage. Automation should be adopted at stages, which consumes most time and effort. This decision essentially depends on the type of product being developed. If a product is conceived with the aim of delivering it once and for all, with no liability for enhancements and maintenance support subsequent to release, then it would be intelligent to invest in automation tools at the unit and integration system testing stages. However, if the product is being developed with the intention of providing sustained support to product recipients and periodic functionality addition thereafter, then the most obvious choice for automation would be the regression testing stage.

Regression testing [4] stands for testing modified code under the same set of inputs used in previous tests. This way we can ensure that modifications in the code did not introduce new errors into the software and verify that modifications successfully eliminated existing errors. It involves executing a pre-defined battery of tests against successive builds of the product to verify that bugs are fixed and functions that were working in the previous build haven't been broken. A more precise definition [5] terms regression testing as equivalency testing, that is, rerunning a suite of tests to assure that the current version behaves identically to the previous version except in such areas as are known to have been changed. Regression testing is an essential part of testing, but is very repetitive and can become tedious when manually executed build after build. Moreover, while running the entire regression test set manually, testers tend to be selective in test case execution due to stringent time constraints, again leaving scope for the final deliverable containing a few unnoticed defects. To further optimize the regression testing, a number of test selection techniques have been proposed by many. [6], [7], [8], [9].

A typical test scenario requires some basic essential steps for an automated black-box testing exercise. These include:

- Test Plan Building
- Test and User Management
- Test Scheduling and Automated Execution
- Result Analysis and Logging
- Report Generation
- Defect Tracking

An ideal test framework should encapsulate all these functions to automate the test management and execution tasks in an effective way. It should also implement a mechanism whereby all test cases of the test plan of the product-under-test can be classified and arranged in the most optimal manner to facilitate organized testing subsequently. Moreover, the framework should be able to store scripts and test parameters and use the appropriate scripts at the time of testing.

4 An Open Framework Approach

Keeping regression testing as the target for automation, this paper describes an open architectural framework for regression testing, which intends to address some of its bottlenecks and irritants. Though the framework specializes in automating the system (black box) testing of networking software, (such as that of devices like routers, switches, integrated access devices and others of similar nature), it is fairly generic in

nature and can be extended for the testing of other software as well. The framework eliminates manual intervention by mimicking the actions of a tester through its scripts. With this framework, the commands or instructions that a tester would otherwise issue manually (for example, through a PC), come from a simulation of the keyboard rather than the keyboard itself. To validate the results of the test, the framework captures a portion of the terminal output, compares it to expected results, and then decides whether the test case succeeded or failed.

Overview

The salient features of the framework are discussed in this section. Essentially, it is designed to provide for black box testing of software (for example, networking software), given the test plan of the product under test. It supports a multi-threaded, multi-user environment and is responsible for maintaining sessions with all the users who have logged on.

It provides a GUI which captures all information related to regression tests such as test script parameters, hardware test bed(s)[1], test plan for one or more builds/releases of the product, users of the system, test schemes[2], etc. At the time of test execution, it allows (through its GUI) users to Start, Stop, Pause or Cancel the current execution of test cases at any given point of time.

It also facilitates the test cases to be easily designed and organized in order to streamline the testing process by grouping optimally the functional features as well as hardware resources required for testing. This is achieved by introducing the concept of a dependency tree in the framework design. This feature gives the users the flexibility to run any test case or a group of cases of their choice, and also serves to utilize the hardware resources optimally. At the end of a test execution run, a number of reports are generated for user reference.

The framework has an interface with advanced automation scripts giving the user complete control over test execution and provides for quick, consistent, reliable and repeatable automated tests. Simulating the presence of a "virtual" user, the scripts execute pre-defined streams of actions, and compare the output to valid states to determine whether or not the test was successful.

Framework Architecture and Components

The framework comprises of two main components, namely:

- Engine component
- Script component

The engine component is the test case manager and test script driver. It is the central controlling entity that controls the concurrent execution of scripts on various test beds and handles multiple simultaneous sessions with testers who are using the framework for testing. The engine further has three sub-components, namely:

---

[1] A group of equipment physically connected in a certain configuration, which is used to test a set of functional features of the product under test.

[2] Those sub-sections of the test plan of the system under test, for which the user wishes to run test cases at a particular instance.

- Engine client (GUI)
- Engine server
- Database repository

The script component consists of intelligent scripts that actually simulate the presence of a user performing a device configuration followed by a test case execution. The scripts also have the ability to determine whether a test case has executed successfully or failed. The scripts, after completion, return the result of the test case to the engine component which is responsible for logging the same in its database and reflecting the result on the GUI so that the tester is constantly updated with the latest status.

The architecture of the framework is depicted in Fig.1. The various sub-components of the engine are discussed in the ensuing sections.

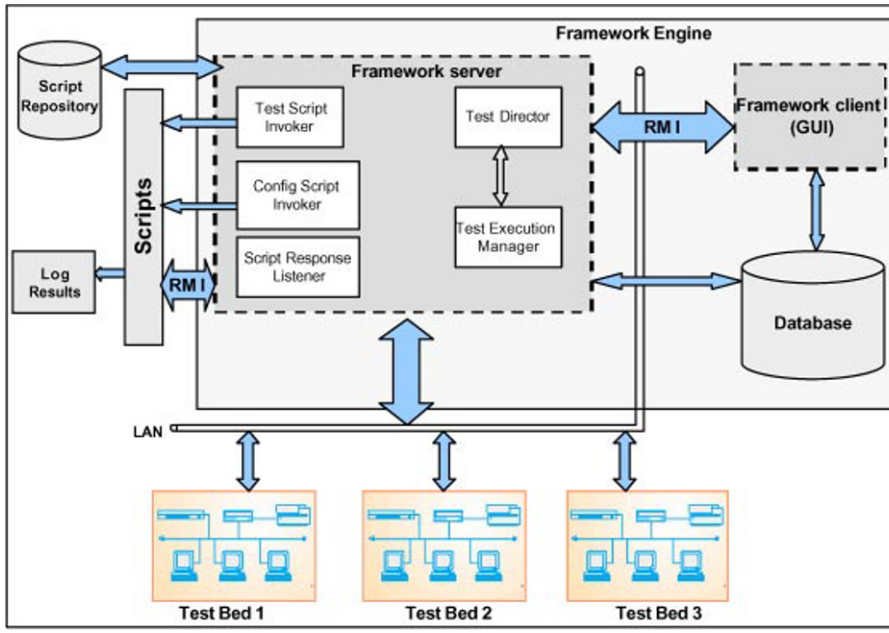

Fig. 1. Framework Architecture

Engine Client (GUI). The engine client provides a convenient interface to the user for entering the data for storage into the framework and for viewing results of the test execution. There are a number of screens, which facilitate this data entry. The kind of information that has to be entered into the framework engine pertains to the release/build of the product-under-test, the details of equipment that constitute a test bed, the number of such test beds, the test plan tree of the product-under-test, test scripts and their expected parameters, etc. If required, more information can be added (such as additional test cases defined for a feature) or existing data updated (such as test scripts, script parameters, etc.) at a later time. Once all the information discussed above is registered with the framework, the users just need to define their individual test schemes and perform tasks such as Start Testing, Stop Testing, and Verify Test Setup for each run of the test cases.

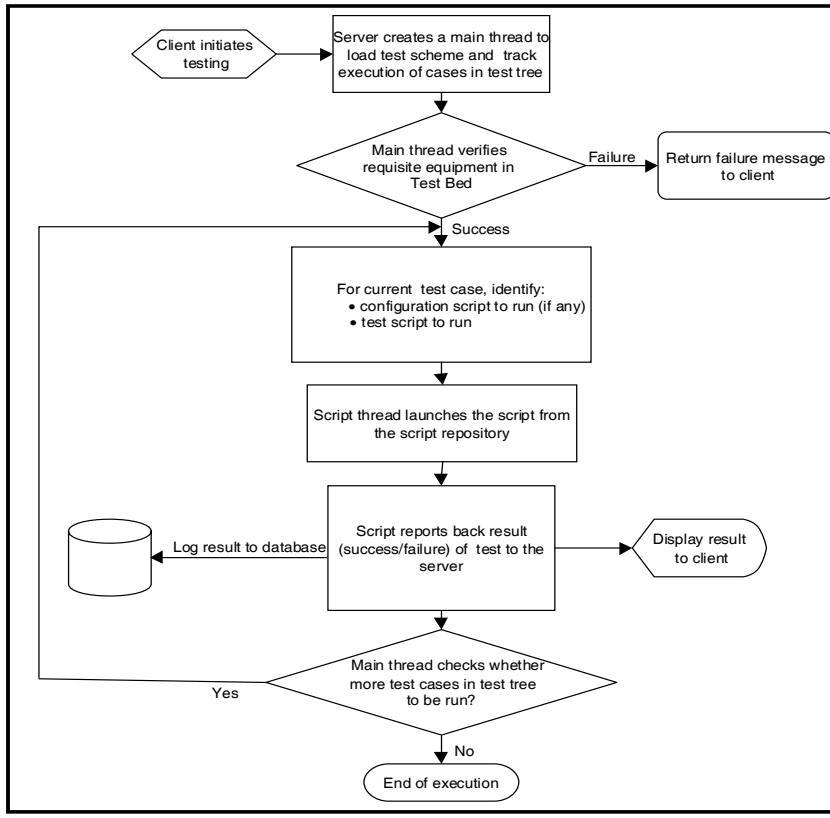

Fig. 2. Task Flow of each Test Execution Request

At the end of a test execution run, progress bars on the main screen give an at-a-glance view of the percentage of the test cases that have succeeded/failed. Detailed results can also be viewed through the GUI itself. A 'History' screen is provided for the purpose of viewing results of any past execution. The results view of historical runs can be filtered by specifying sorting criteria. Summary reports of the test execution results can also be generated through the GUI.

Engine Server. The interaction of the server with other entities, that is, client, database repository and scripts is illustrated in Fig.1 and the task flow of each test execution request as processed by the server is represented in the flowchart in Fig.2. At all times, only one instance of the server runs and it allows multiple clients to connect to it concurrently. The server allocates a unique identifier to each test execution request from a client and tracks that request to completion maintaining this identifier as the key. While multiple clients can execute their schemes concurrently, the execution of test cases within a test scheme is sequential. The server is multi-threaded with a thread allocated to each script, and is responsible for launching the correct script and waiting (for a pre-defined finite time period) for it to return execution results. To maintain the integrity of each client's test execution, the server ensures that each test execution occurs on a separate test bed so as to prevent interference.

Once a script reports its test execution result to the server, the server stores the result in the database, with the unique test execution identifier as the key. Java RMI is used for communication between client, server and scripts. Taking into consideration that all entities in the test bed need not necessarily support the same operating system, the framework has been developed in Java so as to provide platform independence and hence greater flexibility.

Database Repository. The framework uses a Relational Database as a repository for persistent storage of information (that is, products, test plans, test cases, scripts, parameters, test beds, test results). Most of this information is defined in a one-time data entry by the administrator through the GUI and can be reused for later executions. The framework currently provides support for Oracle 8i and MySQL [10] as the underlying databases.

Scripts. The scripts in the framework are primarily written in Expect/TCL, but it also has adapters to some other scripting languages such as Perl, Python, C, etc (Refer Fig.4). Expect [11] is a non-proprietary scripting language, which automates interactive testing by imitating the manual test steps performed by an individual. With the use of Expect, it is possible to simulate a virtual user and perform actions that a user would carry out manually to conduct a test. The scripts in the framework are written based on the test case procedures in the system test plan of the product-under-test. This test plan is pre-defined by the testers and contains all possible test cases to test the various functional features of the product. The scripts are written in such as manner so as to encapsulate the functionality defined by each test case. Thus, the scripts have the intelligence to implement the logic of each functional feature of the product-under-test and of each test case within that feature. The engine controls the execution of the scripts in the correct order and handles the results returned by these scripts.

In case the functionality of the product-under-test undergoes modifications/enhancements, it is likely to impact the test cases for that functionality. Since the scripts corresponding to the changed test cases in turn also get impacted, therefore, it is required to modify the existing scripts or add new scripts in order to reflect the changed functionality.

The new scripts can easily be integrated into the framework engine and subsequently run in an automated fashion as part of the regression test suite for that product.

Hierarchical Test Case Management

A key feature of the framework is the introduction of a dependency tree concept in its design for effective test case organization and management (Fig.3). The grouping of test cases is done with a two-fold objective:

- To group test cases in functional clusters starting at the highest level functionality right down to the test case level, so as to facilitate testers to start testing from any intermediate node in the hierarchy and test all the test cases that lie below that node.
- To group test cases in such a manner that the hardware resources are utilized optimally and clusters are formed so that test cases using similar hardware configurations are placed in the same group. This would reduce manual intervention to a minimum.

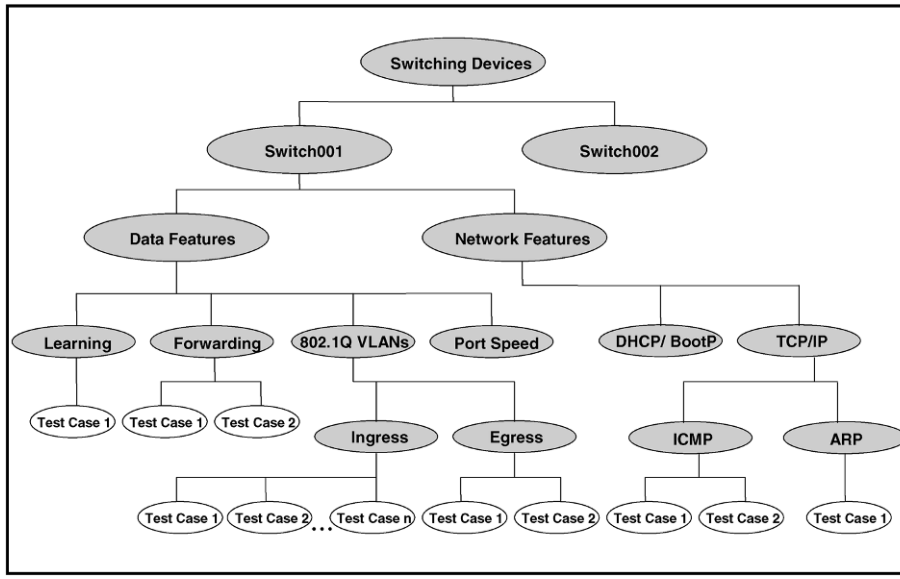

Fig. 3. Hierarchical Test Case Management

The hierarchical test case management approach:

- allows the association of multiple test cases/scenarios to a particular node.
- allows the association of a particular node to a hardware setup. All cases lying below that node in the hierarchy will use the same hardware test setup. For example, if we define a test bed setup at node "Data Features" (Fig.3.), then all branches of the tree lying below this node in the hierarchy would be able to be tested without altering the physical configuration of the defined test bed.
- streamlines the functionality based on scenarios.
- gives the user the ability to perform testing from a particular node/point.
- covers all the test cases under that node.
- bypasses all cases under a node, in case of failure at that node.
- uses the existing structure of the test plan as a baseline, in case a new release is introduced.
- allows testing of a new product, by reusing the entire tree of an existing product for the new product (provided the new product is similar in nature to the existing one and has the same test plan structure).

5 Collaboration with Other Testing Tools

With the growing need for automation, there is a wide spectrum of testing tools available in the market. One of the limitations of most of the testing tool from various vendors is the proprietary scripting language or APIs that come as a package with the tool and hence enforce the binding for a user to use it as it is. Our framework endeavors to do away with this limitation by providing adapters to other tools and scripting languages, so that the strengths of both can be leveraged and synergized to provide an

optimal testing solution. The test case management capabilities of the framework can be integrated with an underlying proprietary or non-proprietary scripting tool, so as to benefit from the advantages provided by both. Envisaging this, adapters have been developed in the framework for integrating with underlying scripting languages such Expect, TCL, Perl, Python and 4Test (A proprietary scripting language of the Silk-Test® Tool [12]).

Moreover, since various tests can be automated best in a particular scripting language, (for example, '4Test' for a GUI automation; 'Expect' for Command Line Automation), the framework can interleave running of these scripts, by invoking the appropriate adapter at runtime. The above feature of collaboration with other tools is illustrated in Fig.4.

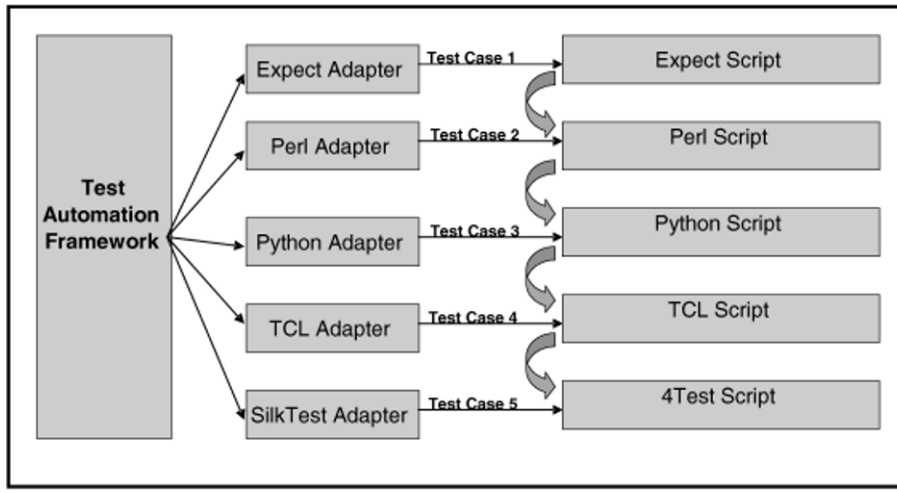

Fig. 4. Collaboration with Testing Tools

6 Framework Deployment

A typical instance of the framework deployment is depicted in Fig.5, with reference to which, the step-wise explanation of how a test execution request is handled by the framework is discussed as follows:

1. The user at the framework client PC chooses a test scheme (that is, test cases to be run in one cycle) from the framework GUI and triggers the testing by pressing the 'Start Testing' button.
2. Once the request reaches the framework engine server, the server extracts information from the database pertaining to:
   - The hardware test bed on which the tests are to be run. (Note: Concurrent testing on more than one test bed is supported by the framework, so it is essential to determine for which test bed a certain execution is targeted.)
   - The scripts corresponding to the test cases in the test scheme.
   - The correct parameters that are to be passed to each script for it to execute the intended functionality of each test case in the test scheme.

3. The framework engine server launches the scripts one by one from the script repository in the correct order onto the hardware test bed. Since the framework is on the same LAN as the test bed, the scripts are able to communicate with all the equipment within the test bed. The scripts are launched in the order in which the test tree is defined in the test scheme. The execution commences with the execution of 'verification' scripts that poll requisite equipment in test bed to ensure their presence for subsequent testing. This is followed by running of 'configuration' scripts that execute the pre-condition to a test case by configuring all the equipment in the test bed with the desired configuration parameter values. Finally, once the configuration is successful, the 'test script' is run. In case the configuration for a particular test case is not successful, then the framework bypasses the test case execution and moves onto the next one.
4. Each script, after completing its execution, reports its result (success/failure), to the framework server. The success or failure result is also accompanied by a brief remark regarding the reason of success/failure.
5. The server logs these results into the database. Simultaneously, the result is updated on the framework client GUI in the form of a progress bar for user reference.
6. Once the entire test scheme has been executed, the same is conveyed to the user by a prompt on the GUI. The user can now choose to view or print the various summary reports, which provide details of the test execution results. The script failures can then be investigated and resolved by the user.

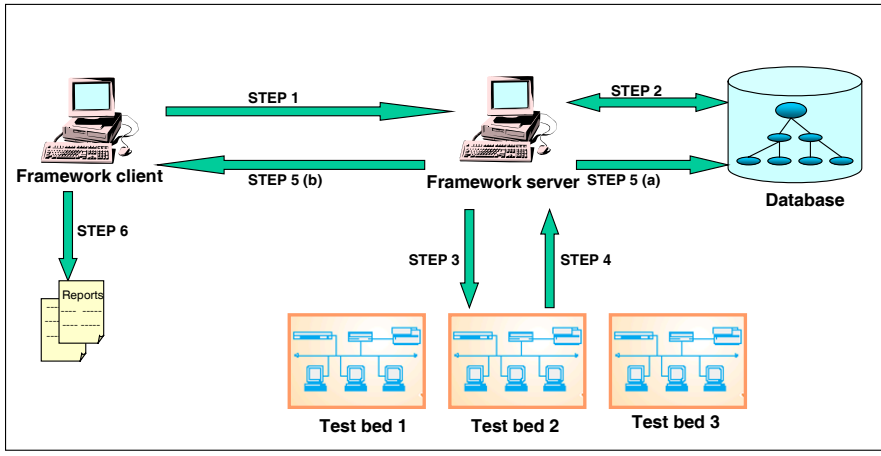

Fig. 5. Framework Deployment Scenario

7 Other Applications of the Framework

Though in this paper, we have taken networking equipment testing as the test scenario, the open architecture of the framework allows it to test a number of other applications and does not limit its ability to testing of networking equipment alone. We have deployed it for the testing of a certain operating system abstraction software, which provided abstraction to an embedded software developer from the details of any underlying real-time operating system. This operating system abstraction soft-

ware [13] was required to be regression tested with each port to a different operating system in order to validate its functionality across various operating systems. The system test cases were to remain the same for regression runs across different operating systems and hence the test automation framework provided a befitting solution for the regression testing of the software.

8 A Comparison with Other Frameworks

The framework under discussion is comparable to some of the others available in the market, intending to do similar tasks. Some such equivalent frameworks include TestDirector from Mercury Interactive [14], STAF (Software Test Automation Framework) [14] and Telcordia™ Mynah System [16].

A brief parallel with each of these is drawn in the following discussion:

TestDirector

TestDirector is Mercury Interactive's software test management tool. TestDirector helps organize and manage various phases of the software testing process, including planning tests, executing tests, and tracking defects. TestDirector is Windows-based and helps create a framework, which acts as a foundation for a testing workflow.

If a parallel is drawn between our framework and TestDirector, it will be observed that they both perform a number of similar tasks, but in a different manner. These include management of test plans and test cases for each product, supporting automated test execution (local or remote) with the help of an underlying scripting language/tool, result logging and report generation for a test set executed, and multi-user support.

TestDirector has additional components like Version Control Manager (for configuration management of test scripts), Defect Tracker (for logging, assigning, fixing and tracking defects to closure), Test Plan import facility (for directly importing test plans from a word document or an Excel spreadsheet), reordering facility (for rearranging test cases before execution), integrated support for WinRunner, [17] (script generation tool from Mercury Interactive, which has its proprietary scripting language-TSL) and third-party tool integration facility through its Visual API.

Our framework has differentiating features like an intelligent test plan tree (for hierarchical management of test plans based on product functionality), facility for specification of configuration and test parameters (which are stored persistently for subsequent regression runs), platform independence (as it works on Windows as well as Linux platforms), and facility to run interleaved tests through different underlying scripting languages. Our framework is focused on the test process of network equipment as against TestDirector, which is a general framework for any enterprise application software testing.

STAF

The Software Testing Automation Framework (STAF) is a framework designed to improve the level of reuse and automation in test cases and test environments. The goal of STAF is to provide a complete end-to-end automation solution for testers.

This framework is open source, and is available under GNU license. STAF can run locally or remotely, provided STAF is installed on the machine. It operates in a peer-to-peer environment; in other words, there is no client-server hierarchy among machines running STAF. A number of processes can be launched simultaneously provided an equal number of resources are available for STAF to be able to handle their concurrent execution. Back-end applications can be developed in Java, C, C++, REXX, Perl and TCL.

When a parallel was drawn between STAF and our framework, it was observed that both support console environment, both have a Java client (for specification of test data and triggering execution) which can work across platforms, both support a multi-user environment, amongst other features.

What STAF does not apparently have is a persistent data storage repository, a report generation facility and managed testing of test scenarios.

All in all, the STAF framework exhibits a lot of resemblance to our framework as is evident from the comparison drawn above and can be considered as its subset as far as its features and capabilities are concerned.

MYNAH

The MYNAH System provides a fast and reliable way to leverage data stored in legacy systems by using scripts to automate functions. It performs the data-intensive work that traditionally required manual intervention. The MYNAH System can be used for:

- Data reconciliation
- High-volume data entry
- Database and system conversions
- Proactive system monitoring
- Rapid application development
- Software testing

Like our framework, the MYNAH System supports TCL scripts at the backend for automation of test cases. It also provides a Graphical User Interface, through which:

- Test scripts can be captured using the automated script builder (which allows scripts to be recorded).
- Test scripts can be modified and enhanced using TCL language package commands.
- Test results can be processed using test management features.

MYNAH System can also be customized for specific testing requirements and apparently supports Unix platform.

9 Return on Investment

In order to arrive at the return on investment achievable with the use of this framework, a pilot was performed to automate the regression test process of an integrated access device (IAD). This device under test integrated voice, data, video and Internet services on up to two broadband connections over TDM and Frame Relay or DSL or ATM. It combined a channel bank, CSU/DSU, IP router, DHCP server, and firewall

in one small footprint. In order to test its features in a black box environment, a number of support equipment such as CSU/DSU, DAC, Router, Abacus (third-party tool for voice generation), and PCs were required to be present in the test bed. Various parameters of these support equipment were required to be configured with different values before actually running individual test cases. A sample of 15 data features of varying complexity was picked for the purpose of the pilot automation. The features of the device that were tested are illustrated in the functional hierarchy in Figure 3.

These were first executed manually to estimate the time taken for one regression test run without any automation aid. These were then organized in a functional hierarchy and automated by writing scripts. They were then integrated into the framework and executed in an automated fashion. The exercise claimed an investment of 3 months, but the benefits derived from it subsequently made it a worthwhile activity. The summary of the manual vs. automated tally obtained at the end of the pilot is presented in the following table.

Table 1. Manual vs. Automated Execution Results

No. of Test Cases in Scope	Manual		Automated	
	Execution Time (hh:mm)	**Elapse Days**	**Execution Time (hh:mm)**	**Elapse Days**
15 data features consisting of 125 test cases	70 hours 45 min.	9 days	16 hours 30 min.	1 day (unattended)

$T_{SE} = [(Time_{manual} - Time_{automated})/ Time_{manual}]* 100 = 76.6\%$
$T_{SD} = [(Elapse_{manual} - Elapse_{automated}) / Elapse_{manual}]* 100 = 89\%$

where:
T_{SE} = Time saving in terms of execution time
T_{SD} = Time saving in terms of elapse days

10 Conclusion

In today's environment of plummeting cycle times, test automation has become an increasingly critical and strategic necessity. It should be adopted as an integral part of the test cycle because few investment opportunities offer a more tangible or reliable return on investment than test automation. A number of frameworks have been conceived with the intention of facilitating test automation at various testing stages. [18], [19], [20], [21]. The previous sections discuss one such framework which attempts to alleviate testers from the tedious task of manual testing.

The framework has been put to test in a real test environment and has proved to be extremely cost-effective in cutting down the test cycle time, as well as meeting other objectives of test automation.

Future challenges lie in enhancing its capabilities to cater to a wider segment of applications/systems to test. In its current state, it can help in the functional/regression test automation of networking equipment, thereby aiding networking equipment ven-

dors in making their test process efficient and easy. Moreover, it can be integrated with some of the other testing tools available in the market.

The usage of the framework for testing more general applications, where a request-response mechanism through a CLI is used to test the application, is being looked at. The framework in such cases would be used to store and manage the test cases, execute the automation scripts in the correct sequence and generate summary reports. Another value addition that is being planned is the building of ready-made test suites for the conformance testing of various protocols.

References

1. Rex Black: Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing. Wiley, John & Sons, Incorporated
2. Edward Kit: Software Testing in the Real World (STRW). Addison-Wesley. (1995).
3. Wallace, D. and Fujii, R.: Software Verification and Validation: An Overview. IEEE Software. (May 1989). 10-17
4. Bret Pettichord: Seven Steps to Test Automation Success. STAR West. (November 1999).
5. Boris Beizer: Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons. (1995).
6. Agrawal, H., Horgan, J., Krauser, E. and London, S.: Incremental regression testing. In Proceedings of the Conference on Software Maintenance (September 1993). 348–357.
7. Chen, Y. -F., Rosenblum, D.S., and Vo, K. -P.: TestTube: A system for selective regression testing. In Proceedings of the 16th International Conference on Software Engineering (ICSE '94), Sorrento, Italy. (May 1994). 211–220.
8. Todd L. Graves, Mary Jean Harrold, Jung-Min Kim and Adam Porter, Gregg Rothermel: An Empirical Study of Regression Test Selection Techniques. ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2. (April 2001). 184–208.
9. John Bible and Gregg Rothermel: A Comparative Study of Coarse- and Fine-Grained Safe Regression Test-Selection Techniques. ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2. (April 2001). 149–183
10. MySQL (http://www.mysql.com/)
11. Expect Home Page. (http://expect.nist.gov/)
12. Segue Solutions (http://www.segue.com/html/s_solutions/s_silktest/s_silktest_toc.htm)
13. mOSAic Design Document , Tata Consultancy Services.
14. TestDirector (http://www-svca.mercuryinteractive.com/products/testdirector/)
15. STAF (http://sourceforge.net/projects/staf/)
16. Mynah System (http://www.telcordia.com/ADAPTX/mynahbft.html)
17. WinRunner (http://www-svca.mercuryinteractive.com/products/winrunner/)
18. Hoffman, D. and Strooper, P.: Automated Module Testing in Prolog. IEEE Transactions on Software Engineering, Vol. 17, No. 9. (September 1991).
19. Chang Liu: Platform-Independent And Tool-Neutral Test Descriptions For Automated Software Testing. In Proceedings of the 22nd international conference on Software Engineering, Limerick, Ireland. (2000). 713 - 715
20. Usha Santhanam: Automating Software Module Testing for FAA Certification. In Proceedings of the annual conference on ACM SIGAda annual international conference (SIGAda 2001), Bloomington, MN. (2001)
21. Schot C.A., Sim M.N. and Kist P.M.: ANT - A Test Harness for the NELSIS CAD System. In Proceedings of the conference on European Design Automation, Congress Centrum Hamburg, Hamburg, Germany. (November 1992)