ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 5, Issue 6, June 2016

Continuous Integration Test Framework

Manoj G R[1], Bhat Geetalaxmi Jayram[2]

Abstract— **Continuous Integration is the practice of testing each change done to your codebase automatically and as early as possible. Continuous Deployment follows the testing that happens during Continuous Integration and pushes changes to a staging or production system. This makes sure a version of your code is accessible at all times. Automation is a cornerstone of a great development workflow. Through testing, you can be sure that the most important steps your customers will take through your system are working, regardless of the changes you make. This gives you the confidence to experiment, implement new features, and ship updates quickly. Automated tests can only show their true power when you run them continuously and for *every* change. If the Continuous Integration infrastructure tells you the tests pass, the tests pass. The advantage of running all tests immediately for every change is that you know right away if something broke. An automated continuous integration test framework is being developed with a simple goal to provide an end to end automated test framework that facilitate optimal testing of multiple flavours of Linux (Host and Guest OS) under different Firmware levels & Operating / Power Hardware environments.**

Index Terms— **Non-LTS, Test Execution, Test Preparation, VM**

I. INTRODUCTION

Continuous Integration is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early. Continuous integration is the practice of merging all developer working copies to a shared main line several times a day. In addition to automated unit tests, organizations using continuous integration typically use a build server to implement continuous processes of applying quality control in general — small pieces of effort, applied frequently. In addition to running the unit and integration tests, such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual quality assurance processes. This continuous application of quality control aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control after completing all development. This is very similar to the original idea of integrating more frequently to make integration easier, only applied to quality assurance processes. In the continuous integration test framework we continuously check for the various releases of Linux which can be either Long Term Support (LTS) or Non-Long Term Support (NON LTS), once we have these releases we download both the iso images and NetBoot files which are later installed on the hardware in following three categories namely bare metal, virtual machine (VM) and kernel based virtual machine (KVM). Bare metal is a direct installation of OS on the Hardware and the other two VM and KVM are the virtual environments of which the former is software level virtualization and later is a hardware level virtualization. Once we have this OS on the system and the system is up and running we perform integration testing to ensure that all test cases are without any errors and accepted by the end user of the systems. All these activities are integrated into Jenkins server for deployment and start executing automatically when there is a release for particular Linux release and generate results which are displayed to the user and a mail will be sent to the Jenkins mailing list.

Manoj G R, *Department of Information Science & Engineering, The National Institute of Engineering, Mysuru, India*,

Bhat Geetalaxmi Jayram, *Associate Professor, Department of Information Science & Engineering, The National Institute of Engineering, Mysuru, India*,

II. LITERATURE SURVEY

A. *Related Work*

Before the Automation, testing was done manually which involved lot of test cases execution and would take lot of amount of time for execution of these test cases and sometimes even two to three days of time. And even the test cases are executed one after the other i.e. serially. These Serial Execution has some of the disadvantages like

- More time for execution of test case as test cases need to be executed one after the other
- No Proper utilization of resources

Because of these drawbacks we move to make automated test framework, we use Python as the chief driving programming language as it has all the features of object oriented approach with additional data structures like lists, tuples and additional libraries that will be handy during the programming. In software testing, test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes. Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or add additional testing that would be difficult to perform manually. Some software testing tasks, such as extensive low-level interface integration testing, can be laborious and time consuming to do manually. In addition, a manual approach might not always be effective in finding certain classes of defects. Test automation offers a possibility to perform these types of testing effectively. Once automated tests have been developed, they can be run quickly and repeatedly. Many times, this can be a cost-effective method for integration testing of software products that have a long maintenance life. Even minor patches over the lifetime of the application can cause existing features to break which were working at an earlier point in time.

Test automation tools can be expensive, and are usually employed in combination with manual testing. Test automation can be made cost-effective in the long term, especially when used repeatedly in iteration testing. In automated testing the test engineer or software quality assurance person must have software coding ability, since the test cases are written in the form of source code which, when run, produce output according to the assertions that are a part of it. One way to generate test cases automatically is model-based testing through use of a model of the system for test case generation, but research continues into a variety of alternative methodologies for doing so. In some cases, the model-based approach enables non-technical users to create automated business test cases in plain English so that no programming of any kind is needed in order to configure them for multiple operating systems, browsers, and smart devices. What to automate, when to automate, or even whether one really needs automation are crucial decisions which the testing (or development) team must make. Selecting the correct features of the product for automation largely determines the success of the automation. Automating unstable features or features that are undergoing changes should be avoided. We are making an effort from scratch to build an automated continuous integration framework. We are reusing some of the pieces available in open source community and putting them together. Autotest is a framework for fully automated testing. It is designed primarily to test the Linux kernel, though it is useful for many other functions such as qualifying new hardware. It's an open – source project, it is used and developed by a number of organisations, including Google, IBM, Red Hat, and many others.

B. Framework approach in automation

A test automation framework is an integrated system that sets the rules of automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. These components act as small building blocks which need to be assembled to represent a business process. The framework provides the basis of test automation and simplifies the automation effort. The main advantage of a framework of assumptions, concepts and tools that provide support for automated software testing is the low cost for maintenance. If there is change to any test case then only the test case file needs to be updated and the driver script and start-up will remain the same. Ideally, there is no need to update the scripts in case of changes to the application. Choosing the right framework/scripting technique helps in maintaining lower costs. The costs associated with test scripting are due to development and maintenance efforts. The approach of scripting used during test automation has effect on costs.

Various framework/scripting techniques are generally used [1]:

1. Linear (procedural code, possibly generated by tools like those that use record and playback)
2. Structured (uses control structures - typically 'if-else', 'switch', 'for', 'while' conditions/ statements)
3. Data-driven (data is persisted outside of tests in a database, spreadsheet, or other mechanism)
4. Keyword-driven
5. Hybrid (two or more of the patterns above are used)
6. Agile automation framework

As new functionalities are added to the system under test with successive cycles, automation scripts need to be added, reviewed and maintained for each release cycle. Maintenance becomes necessary to improve effectiveness of automation scripts. [2] The Testing framework is responsible for:

1. Defining the format in which to express expectations.
2. Creating a mechanism to hook into or drive the application under test.
3. Executing the tests.
4. Reporting results.

III. SYSTEM ANALYSIS

A. *Problem Statement*

Today Power Systems support following Linux distributions, Red Hat Enterprise Linux (RHEL), Ubuntu, and SUSE Linux Enterprise Server (SLES). These linux flavors are supported in various operating environments such as KVM guest, VM guest and also running in non-virtualized environment (bare-metal mode).
Linux on Power test teams are facing a problem of plenty:

- Exponentially growing Hardware & Software combination.
- Expanding Hardware platforms.
- Exploding combinations of Hardware, Firmware and Software.

In addition the test teams are required to frequently validate several generally available Linux distributions on newer platforms and hardware.

B. *Proposed System*

An automated continuous integration framework is being developed with a simple goal to provide end to end automated continuous integration framework to facilitate optimal testing of multiple flavors of Linux (Host and Guest OS) under different Firmware levels & Operating / Power Hardware environments. By using python we develop the continuous integration test framework which is capable of testing certain features on hardware under various different combinations of hardware and software platforms. First we check for the availability of the build from various linux distributions, i.e. Ubuntu, RHEL and SLES. Once the builds are available we check for the availability of the system and install these operating systems under different operating environments. Once the system is booted with any of the linux releases under any operating environments we use the autotest framework which has built in test cases that can be modified and made use for testing the features on the machine. Later after the execution the results can be published on the web based dashboard for further evaluation and analysis.

IV. SYSTEM DESIGN

Large systems are always disintegrated into smaller sub-systems that provide some correlated set of services. The early design process of recognizing these sub-systems and establishing a framework for sub-system control and communication is called Architecture design and the yield of this design process is an explanation of the software architecture. The architectural design process is apprehensive with establishing an elementary structural framework for a system. It involves ascertaining the major components of the system and communications between these components. The system architecture shows the blocks required and is as shown in Fig. 1,

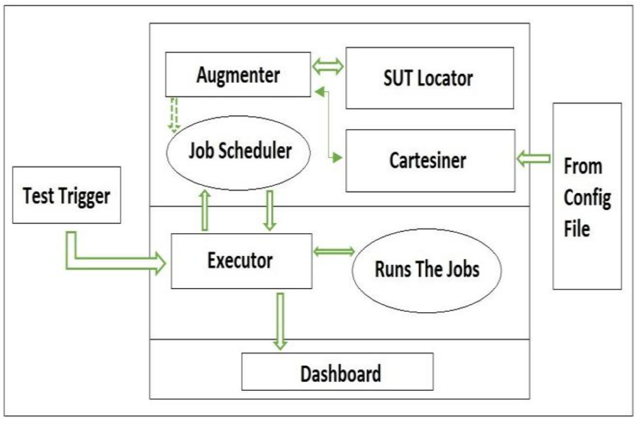

Fig 1: Architecture of Continuous Integration Test Framework

1. First get the build and populate into software repository.
2. Once the builds are available, run them on the system as a bare metal or VM guest or KVM guest and this step would trigger the test cases.
3. Later execute the required test cases to test certain features whether the system is compatible with the various Linux flavours i.e. RHEL, SLES or Ubuntu.
4. As shown in Fig. 1, after the test is triggered the next step is the execution, for the execution of a test it has to have some information as arguments.
5. The Augmenter will have the details about the SUT and this detail is sent to the Augmenter from the SUT locator.
6. The Cartesianer contains details about the IP address, port number, log-in and log-out timeout which is fetched from the configuration file.
7. The combined details at the Augmenter is then fetched by the Job Scheduler which schedules the job in order to execute the test cases which has been triggered. The Jobs will be scheduled depending on the CPU, Memory and I/0 resources available.
8. Then the Job runs and the results would be sent to the executor.
9. Executor publishes the test execution results on a dashboard for analysis.

10. All the above steps are automated and the results of the test cases would be available in the dashboard.

V. IMPLEMENTATION

The main focus of the implementation is to realize the requirements and design into the software code which can be effectively used for the specified purpose. The implementation phase involves actual materialization of ideas, which are expressed in the analysis document and developed in the design phase.

A. *Structure Chart for framework*

Structure chart shows the control flow among the modules in the system, explains the identified modules and the interaction between the modules. As shown in Fig. 2, it also explains the identified sub modules.

The various key components identified are:

1. Build Availability
2. Test Preparation
3. Test Execution
4. Results

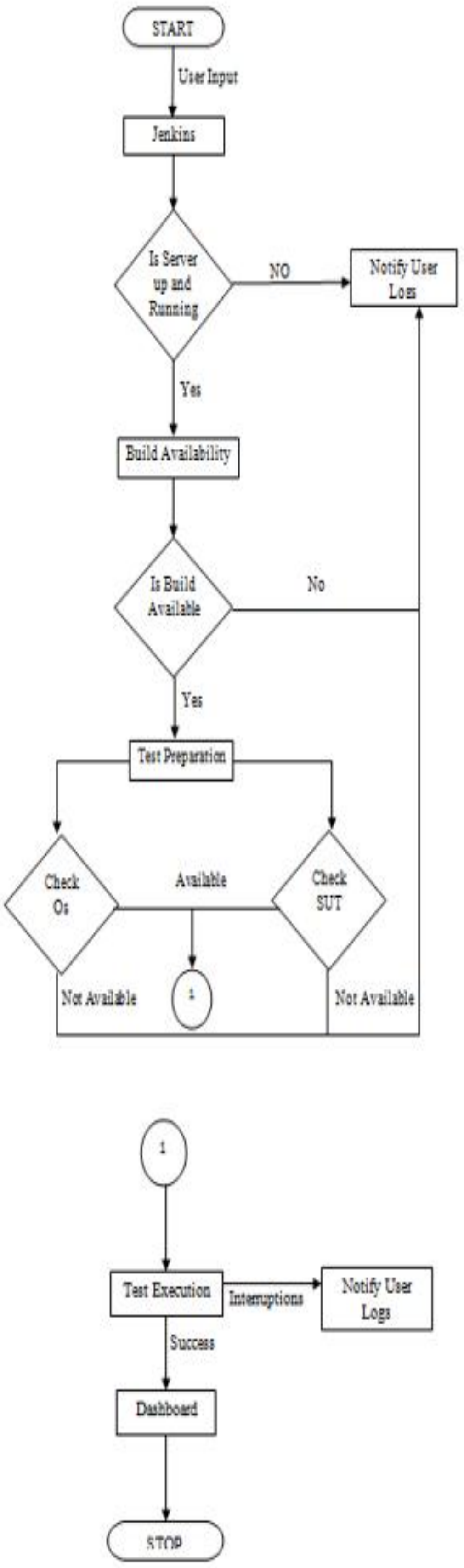

Fig 2: Flow Diagram for framework

1. Build Availability

1) *Purpose:* The purpose of this module is to check for the availability of a particular linux release. The

linux release can be LTS release or the Non LTS which are Long Term Support and Non Long Term Support respectively. Once the build is available the particular release will be downloaded and placed in a repository. It checks for the updates in the release for every week.

2) *Functionality:* Enables user to potentially layer the functionality of the application and give the freedom to the user to specify what linux flavour to be downloaded, i.e. Rhel, Sles, and Ubuntu
3) *Input:* The input to this module can be any flavour of linux that need to be downloaded. They can even specify whether to download iso image or the netboot files or both.
4) *Output:* The output of this module will be a linux release that will be downloaded in a particular repository in a format specified by the user

2. Test Preparation

1) *Purpose:* The purpose of this module is to check for the availability of the system for OS installation and generate optimal test combinations to be executed on the system.
2) *Functionality:* It first generates possible test cases with test cartesianer and finalize optimal test combination with test augmenter and find the system for OS installation by SUT locator.
3) *Input:* The input to this module is the linux release, firmware details and the list of systems that are available for installation.
4) *Output:* The output of this module will be a optimal combination of the tests that are to be executed

3. Test Execution

1) *Purpose:* The purpose of this module is to deploy necessary software and start executing the tests.
2) *Functionality:* It has a module for deploying the required software and get the system for execution and begins execution.
3) *Input:* The input for this module is an optimized test combinations.
4) *Output:* The output of this module will be the result of execution of the test cases which will be even logged to user for analysis.

4. Dashboard

1) *Purpose:* The purpose of this module is to display the execution results for the user.
2) *Functionality:* It has a module to take user input for the test environment and display the results after execution.
3) *Input:* The input for this module is a list of arguments that specify the test environment.
4) *Output:* It displays the execution results to the user.

VI. CONCLUSION

An end to end continuous integration test framework has been developed to perform optimal testing of various combinations of hardware and software. The execution time is considerably reduced as the test cases are being executed in an automated way. As most of the things are automated it includes less man power and execution hours there by reducing the overall maintenance cost. As the framework is being integrated with Jenkins there by making the best usage of available resources. It is also made sure that each and every output of a particular instruction that is being executed is proper logged on to user logs for further analysis and feedbacks.

VII. FUTURE ENHANCEMENTS

Currently the continuous integration test has been implemented as a serial execution of test buckets, the jobs are submitted to the test framework sequentially i.e one after the other due to which the execution time of these test buckets is more. As future work, to enable faster completion of test buckets we can enhance it to support parallel job submission to the test framework. The test execution on multiple systems can be done by parallel execution of the tests there by having an upper hand on serial execution, where parallel execution takes considerably less time for execution.

REFERENCES

[1] https://en.wikipedia.org/wiki/Test_automation
[2] http://www.guru99.com/automation-testing.html
[3] EH Kim, JC Na, SM Ryoo, “Test Automation Framework for Implementing Continuous Integration” Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference, 2009, pp: 784-789.