AUTOPLUS: A GENERIC AUTOMATION TEST FRAMEWORK FOR TESTING AND INTEGRATION CHALLENGES OF COMPLEX/MULTILAYERED SOFTWARE SYSTEM

Amit Kaul, Dr. Priyanka Sharma

ABSTRACT

Testing is an equally important and critical phase as analysis and design for successful deployment of any software. Traditional testing and integration have been manual and time consuming. Concept of automation of testing process has been around for a while and has been deployed and successfully used. But as complexity of software systems began to increase and as organizations preferred to impose more processing on software, the traditional automation was no longer sufficient. Automation Framework concept was introduced for the integration and testing of these systems and many off-the-shelf frameworks were introduced. These frameworks, however, have not been able to provide the automation solutions as they promise. More often than not, most organizations have to use these frameworks as just another tool in the plethora of tools they have purchased and also rely on internal resources to achieve the desired results. Additionally, most organizations have proprietary protocol implementations as well as in-house tools that are widely used by developers and system testers to validate the software.

What we are discussing in this paper is how organizations can efficiently create an Automation Test Framework (ATF) using the combination of off-the-shelf tools, in-house tools, General Purpose License (GPL) tools and various high level scripting languages. The proposed ATF consists of a User Interface, Driver Application, Execution Engine, Unit Under Test (UUT), Verification Engine and a Reporting Engine. The case study describes the implementation of such an Automation Framework for a multi-layered multimedia delivery software system.

Keywords: ATF, Automation, Frameworks, Integration, Testing

1. INTRODUCTION

System Development Life Cycle (SDLC) of any software system consists of system analysis, design, implementation and finally testing. Traditionally the focus has mostly been on analysis design and implementation phases. Software testing is also a critical element of software quality assurance and represents the ultimate review of specification, design and code generation [8].Testing, integration, verification and validation of any software system are most resource intensive and critical phases of the software delivery process and for achieving software quality. It involves the execution of a deterministic software system with test data and a comparison of the results with the expected output, which must satisfy user's requirements which accounts for over 25 % of the cost of a software development [9].

Traditional testing and integration have been manual and time consuming. According to NIST (Department of Commerce's National Institute of Standards and Technology (NIST)) report, software bugs are so prevalent and so harmful that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product. The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. Software developers spend approximately 80 percent of development costs on identifying and correcting defects [14]. Also, the increasing complexity of software, along with a decreasing average product life expectancy, has increased the economic costs of errors. The tragic impacts of some failures are well-known. For example, a software failure interrupted the New York Mercantile Exchange and telephone service to several East Coast cities in February 1998[14]. Automation has always been proposed as the solution for saving time and increasing efficiency of the testing phase. Where Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. Commonly, test automation involves automating a manual process already in place that uses a formalized testing process [12]. In addition, because testing involves running the system being tested under a variety of configurations and circumstances, automation of execution-related activities offers another potential source of savings in the testing process [9]. This Concept of automation of testing process has been around for a while and has been deployed and successfully used. Using automated software testing (AST), developers and software testers can optimize the software testing lifecycle and thus reduce total time consumption [3]. Companies that opt for Test Automation pass the break-even point for labor cost after just 2 to 3 runs of automated test [9].

Along with Automation, Framework concept was introduced for the integration and testing of these systems and many off-the-shelf frameworks were introduced. These frameworks, however, have not been able to provide the automation solutions as they promise. More often than not, most organizations have to use these frameworks as just another tool in the plethora of tools they have purchased and also rely on internal resources to achieve the desired results. Additionally, most organizations have proprietary protocol implementations as well as internal tools that are in-house and widely used by developers and system testers to validate the software. As the complexity of software systems began to increase and as organizations preferred to impose more processing on

software, the traditional automation was no longer sufficient. Some of the major testing issues that are challenges for any software project are:

- Increasingly complex functionalities pushed to software layers.
- Stack of layers, with each layer containing multiple software artifacts.
- Unit, integration, sanity and system testing (Automated preferably).
- These layers/ artifacts communicate among each other via mechanisms like IPC/ RPC/ Soap/ XML/ HTTP etc.

2. AUTOMATED TEST FRAMEWORKS (ATF)

Automated Test Framework is name given to a collection of tools, simulators, concepts, and UUT (Unit Under Test) dependencies that, with system specific customization, provide automated software component validation and testing facilities [12]. The framework provides an entirely different approach, although it is often used in conjunction with one or more data-driven testing strategies [4].More specifically it is a test platform which is flexible and maintainable test environment , supports continuous integration testing, easy to use, configurable execution and verification aspects, aid developers to verify their component prior to system integration and testing, should be scalable etc. What it means is that a Automated Test Framework can be created for most of the complex Internet Protocol (IP) based software products that have component architecture and depends on inter-component communication at peer level or layered level, for the system to work. Any multi-layered Software System is a candidate for developing such Automated Test Framework. Figure 1 shows conceptual diagram of ATF target system.

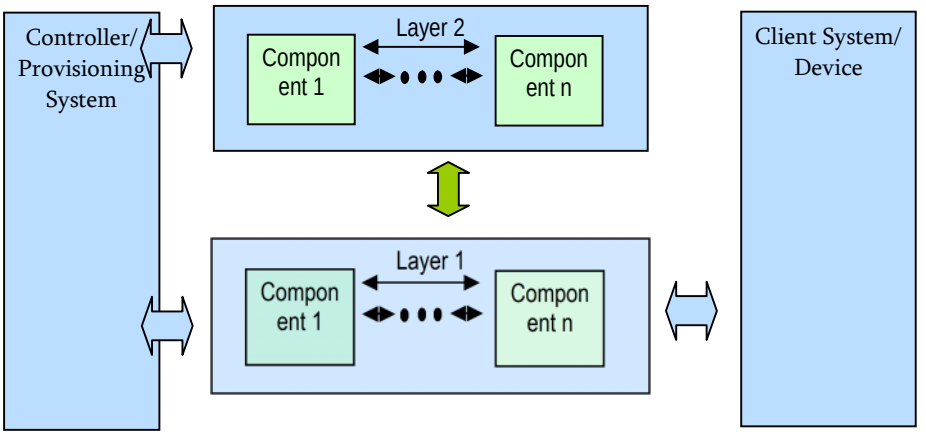

Fig. 1. ATF: Target Systems

Multiple IPC communication protocols support:

- Every organization has proprietary and customized way of designing and building the systems.
- No standardized off-the-shelf tools to suit all organizations (Wishful thinking).
- Specification changes/ market based changes impose heavy toll on resources and deadlines.
- Need for flexible and scalable automated testing methods and tools.
- Shorter release cycles and simultaneous work on, past, current and future features lead to quality consistency and stability issues.
- Automation is inevitable, but how to and what to?

Approach to Automation: It is assumed that the user knows what test automation is all about and he has a planned approach for it. The main ten steps in test automation are as [9]:

Step 1 - Identify Testing Scope
Step 2 - Identify Testing Types.
Step 3 – Identify Requirements to be automated
Step 4- Evaluate Test Automation Tool.
Step 5- Identify Requirements that can be automated
Step 6 - Design Test Automation Framework
Step 7 – Design Data Input Store
Step 8 - Develop framework
Step 9 – Populate Input Data Store
Step 10 - Configure Schedulers

There are many factors responsible for the success of building a Test Automation Framework. The key elements are management commitment, cost and budget, precise definition of actual process and proper resources.

ATF development involves various challenges that include [9]:

- Clear vision of what needs to be achieved out of automation. It should address core questions like testing model, types of testing, which areas need to be automated etc.
- Tool identification and Recommendations process. It includes creating standard tool evaluation checklist, types of testing, and acquiring multiple tools to perform different types of testing.
- Framework design which involves identifying requirements from multiple areas like identification of necessary utility/components, types

of input data store to be communicated, communication between the systems and utility/component development etc.

Present solutions are available but have multiple limitations like:

- No standardized off-the-shelf tools to suit all
- Myriad automation and regression test tools, work perfectly in demos on demo applications, but such "perfection" not achieved with real software.
- Most tools address a specific area of testing – UI, configuration, communication etc.
- To be effective these tools put coding (scripting) pressures on testers.
- Platform dependence.
- Human Resource dependence
- The desired solution should offer:
- Automated execution - Time/event triggered execution and reporting.
- Start at basic layer of the stack and offer scalable framework as more artifacts are added.
- Support automated building, installation and configuration of artifacts.
- Support multiple communication protocols.
- Encompass existing tools and skills within the organization.
- Provide means for developers to perform integration testing before releasing the component.
- Provide means to support easy user driven automation test case development.
- Support sanity, regression and release testing.
- Leverage on exiting tools/ resources within the organization.

AUTOPLUS - Generic Automation Test Framework for Testing and Integration presented ahead attempts to uphold all these promises. It shows how organizations can efficiently create an ATF using the combination of off-the-shelf tools, in-house tools, GPL tools and various high level scripting languages.

3. AUTOPLUS

The main components of proposed ATF known as **AUTOPLUS** are User Interface, Driver Application, Execution Engine, Simulators, Communication Mechanism, Unit/ Layer under test, Verification engine and Reporting Engine as shown in Figure 2.

User Interface (UI): It is a web service that allows building, executing, monitoring and reporting management services with features like Easy and intuitive UI, Multiple Version Control System support, Triggering mechanism and chaining mechanism.

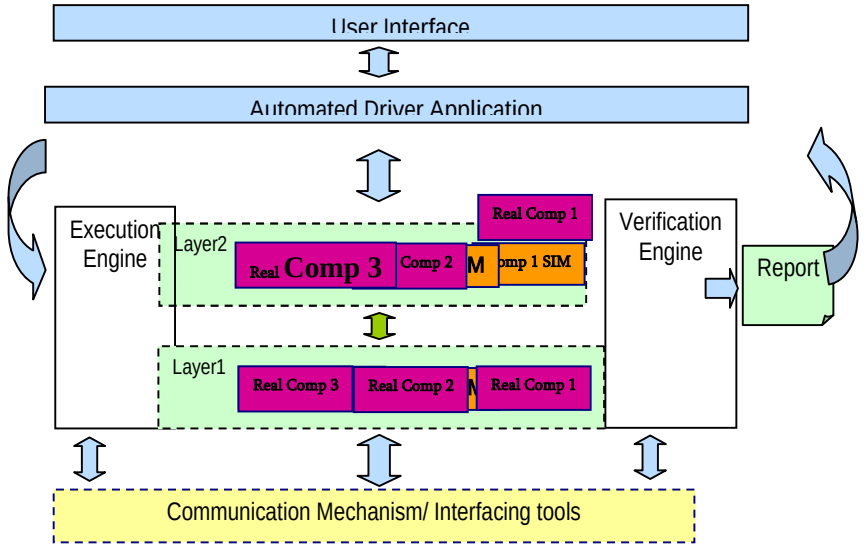

Fig. 2. AUTOPLUS Architecture

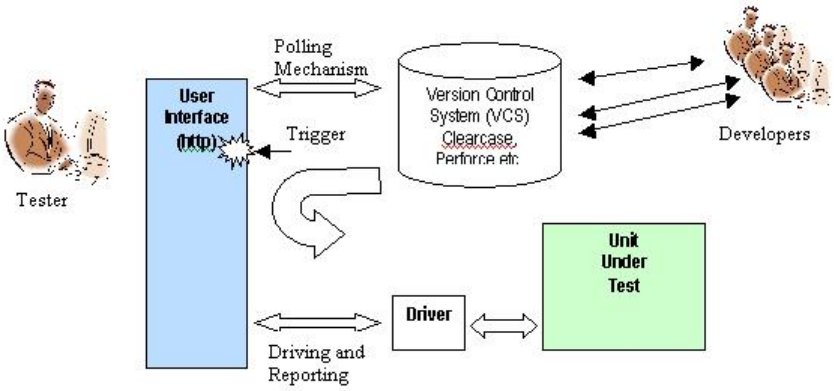

Fig. 3 : AUTOPLUS User Interface working mechanism

As shown in Figure 3, the User Interface has to perform multiple functions apart from being access mechanism for the user. In any multi user and multi feature development environment, the developers will constantly be working on new features and would be moving their unit-tested code into source control. The

User Interface shall has a mechanism to keep on polling the Version Control System (VCS) to determine and record all the changes in the project of interest since last test was run. This also helps in keeping track of who made what change in the system. The User Interface has a built-in triggering mechanism that starts the whole test process. The triggering mechanism can be time based (predefined time specified while setting up the system) or an event based (A particular file checked in, a file generated, some other task completed etc.). Once the trigger happens the UI refreshes all the files from the VCS and then starts the driver application with appropriate options as configured by the user. Driver Application is described next. Once the system is up and running, the UI constantly collects and displays the status of the process being executed as well as over all status. Another feature of the UI is the Chaining Feature. Chaining means triggering build/ test processes sequentially or in parallel. While a build of a component is going on, build or test of another component can be started. If a component has requirement that some other components are build before it is built, then the build chain can be made sequential.
Examples of tools that can be used are Zephyr, Team City etc.

Driver: A software artifact that provides facilities to drive execution of the automation process and manage the order of sub processes within the automated process. It can easily add, remove and modify the process.

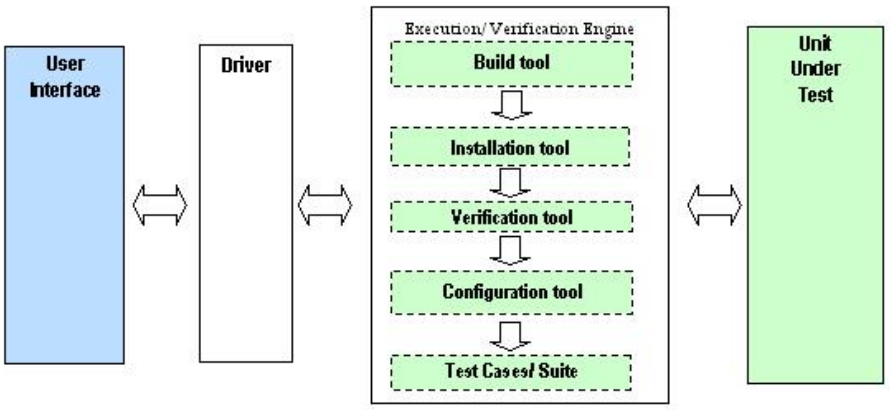

Fig. 4. AUTOPLUS Driver/ Execution Engine working mechanism

As shown in Figure 4, the Driver tool invokes sequence of tools in an automated fashion within an execution engine framework, once triggered from the User Interface. The Driver is the component that holds the logic for ordering and timing of invoking the sub tools within the execution engine, which is described next. The Driver should treat each on the tools within execution engine as 'Projects'. The tools contained within the execution engine may be command

line driven, script driven or some other mechanism. The tool should provide mechanism to easily configure the invoking mechanism of each of these tools independently. In addition, the order of execution as well as timing constraints should also be taken care of. The Driver shall also be flexible enough to support invoking or external applications, if need be, from the automation process.
A popular tool that can be used as driver is Ant. It provides simple xml based interface for configuration.

Execution/ Verification Engine: It is collection of software tools that perform the job of configuring and setting up the system as well as verifying the setup and finally executing the test scenarios on the system. It has features like script based configuration, pre-packaged well defined input configuration data that is used for verification at the output. The sequence of execution is as follows

1. First the build process of the artifacts or components that are to be tested is started by the 'Build Tool'. The Build Tool can typically be a perl or python script that uses the existing build mechanisms, as are being used by the developers. For instance it can invoke the make files, for C++ components, or java utilities for java components. The tool will then copy the components thus built into a predefined location.
2. The 'Installation Tool' performs installation of the components build in earlier step. This can again be a set of simple scripts that copy the components to relevant locations on the unit under test and also start them. It can be configured to pick up real components when they are released by the developer and use component simulators till then. As more and more components are released, the installation tool can be configured dynamically to replace simulator by the real component.
3. Following the installation, the 'Verification tool' is executed. This tool can be written in any high level language and it performs the task of verifying that all the components started in step 2 are actually up and running and are enabled.
4. The next step is configuration of the system. The 'Configuration Tool' configures or provisions the system for actual usage. It configures parameters on various components running on the system and uses predetermined and well known parameters to configure the system. The advantage of using a well known parameter set is that there are some values against which the test cases can be executed during the final stage. Many off the shelf tools like SoapUI are available to perform this job.
5. Finally, the Driver application executes the test cases or a Test Suite, which perform the validation that the system is working. The test cases can again be written in a high level language. The test suites typically will have

underlying libraries that fetch talk with different components on the system and extract the values to be compared in the test case.

Simulators: Simulator is collection of software artifacts that provide simulation of components above and (or) below the unit under test. This is a dynamic set of tools that are increased or reduced in number based on which components are being put under test. It has features like Simple simulators (Clients/ Servers), that perform the designated duties of the real component with regard to the UUT.

The concept is that at the time of startup, simulators are available for most of the components. Suppose Real Component 1 is ready for testing. The developer can easily configure the AUTOPLUS process to install his real component on the UUT. Once the system is setup, all other required components will still be simulators. Thus a developer can test his/ her component without waiting for other components to be released. As time passes, other simulators can be replaced by their real counterparts until all the real components are in use. This can help in dramatically reducing the number of integration bugs, feature bugs as well as programming bugs even before the system is ready to be tested by the system testing group.

Also, as it is based on prepackaged well defined input data, it allows customization of parameters per test and 'pushing'/ 'pulling' of configuration data per scenario. Component developers often develop local simulators for unit testing; these simulators can be leveraged upon and used in the AUTOPLUS.

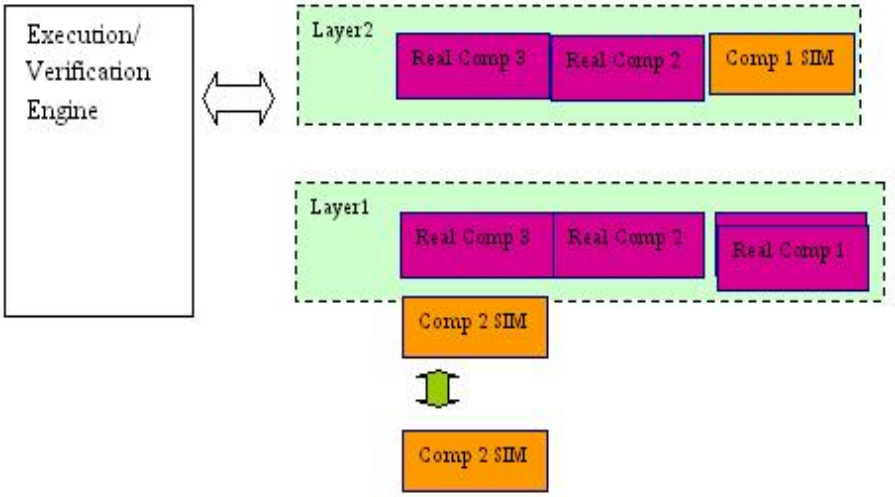

Fig. 5 . Unit Under Test (UUT)

Communication Mechanism: A unified communication mechanism that encapsulates the component specific activities like start, stop, restart etc. under a

unified interface and provides a standard interface to all other parts of the framework. Also, it

- Hides individual complexities
- Provides consistent unified interface to control/ manipulate the component and simulators.
- Provides API's to execution layer and verification layer.
- Might include interface to the standard messaging mechanisms as required for inter-component communication.
- Facilities to read logs from any of the artifact in the system.

An example of unified communication mechanism is the JMS (Java Messaging System). This provides a mechanism for components to interact with each other with an assurance that the message will be delivered to the peer. Another example is a standardized Socket communication mechanism for data as well as configuration message exchange between components. Modern day software systems use standard data exchange standards like XML and SOAP to communicate among components, for these many cheap and open source tools are available for using them for exchanging data.

Unit/ Layer under test: As shown in Figure 5, in a multi layered software system the whole system is a Unit Under Test. In AUTOPLUS concept, however, each unit within the system can be treated as a UUT. In Figure 5 for instance, Layer 1 can be a UUT and Layer 2 can be a simulated or mock layer. Once Layer 1 is done with and Layer 2 is ready for testing, Both Layer 1 and Layer 2 become UUT. If only few components are ready from layer 2 then rest will still be Simulators. Thus UUT may then be Layer 1 plus 2 components of Layer 2.

REFERENCES:

1. C. Rankin, The Software Testing Automation Framework Software Testing and Verification, IBM system journal, Volume 41, Number 1, 2002
2. Daniel Mosley, Bruce A Posey ,Just Enough Software Test Automation Prentice Hall ,2002
3. Elfriede Dustin, Thom Garrett, Bernie Gauf, Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality , Addison-Wesley Professional, March 2009
4. Kaner, C., "Pitfalls and Strategies in Automated Testing" IEEE Computer, April, 1997
5. Mark Fewster , Dorothy Graham ,Software Test Automation , ACM Press, 2000
6. N Gnanasekaran, Vineetbanga Automated ,"Regression Testing Framework", NIIT Technologies Ltd July 2007
7. Rick Mugridge, Ward Cunningham, Fit for Developing Software: Framework for Integrated Tests, Jun 29, 2005, Prentice Hall
8. Roger S. Pressman, Software Engineering – A practitioner's approach, McGraw-Hill International Edition, 5th Edition,2002.

9. http://www.articlesbase.com
10. http://www.automatedtestinginstitute.com
11. http://citeseer.ist.psu.edu
12. http://en.wikipedia.org
13. http://www.ibm.com
14. http://www.nist.gov
15. http://www.netbsd.org
16. http://www.softwaretestinginstitute.com
17. http://www.testingfaqs.org