Usage of Robot Framework in Automation of Functional Test Regression

Article · October 2011

CITATIONS
9

READS
9,592

2 authors:

Stanislav Stresnjak
Siemens Convergence Creators
2 PUBLICATIONS **24** CITATIONS
SEE PROFILE

Zeljko Hocenski
University of Osijek
93 PUBLICATIONS **918** CITATIONS
SEE PROFILE

Usage of Robot Framework in Automation of Functional Test Regression

Stanislav Stresnjak
Siemens CMT
Osijek, Croatia
e-mail: stanislav.stresnjak@siemens.com

Zeljko Hocenski
Computer and Software Engineering Department
University Josip Juraj Strossmayer in Osijek
Osijek, Croatia
e-mail: zeljko.hocenski@etfos.hr

Abstract — **Manual testing is a time consuming process. In addition, regression testing, because of its repetitive nature, is error-prone, so automation is highly desirable. Robot Framework is simple, yet powerful and easily extensible tool which utilizes the keyword driven testing approach. Easy to use tabular syntax enables creating test cases in a uniform way. Ability to create reusable high-level keywords from existing keyword ensures easy extensibility and reusability. Simple library API, for creating customized test libraries in Python or Java, is available, while command line interface and XML based output files ease integration into existing build infrastructure, for example continuous integration systems. All these features ensure that Robot Framework can be quickly used to automate test cases. This paper describes how it is used for automation of existing functional regression test cases within short time and with great success and thus saving costs and enhancing the quality of the software project.**

Keywords-software testing; integration testing; regression testing; test automation; robot framework

I. INTRODUCTION

In order to integrate a component within a larger system, three major properties, the fitness, the correctness, and the robustness, have to be tested [1]. The fitness of a component for an application is in general treated as the compatibility of the provided interface of the component and the specification of the required interface of the application. The correctness of a component is its ability to return the correct output when provided with the correct input, while the robustness concerns the absence of a behavior possibly jeopardizing the rest of the system, especially under wrong input. When lot of components is present, integration testing became quite complex and one of the software development improvement steps pertains to testing process improvements which can hardly be done without test automation.

There are various tools for test automation available – commercial and open source, but few are suitable for black box testing (for a black-box testing, see [2]). Many of available tools are most suitable for the unit tests performed by the developers. When it comes to the integration testing or functional verification – not so many tools are available.

Many of the testing tools provided by vendors are very sophisticated and use existing or proprietary coding languages. Effort to automate existing manual tests is similar to a programmer, using a coding language, writing program in order to automate any other manual process [3].

This paper is organized as follows. Section 2 explains how the tool choosing is done. Section 3 describes why specific tool was chosen. Section 4 describes the implementation of the tool. Section 5 is about benefits of the automation. Section 6 draws conclusions.

II. CHOOSING THE TOOL

What was needed was a tool simple enough to make fast automation and in the same time powerful so these tests can be extended and produce less error prone. The tool should be platform independent. Client tests were run on Linux and Windows and server tests were run on Linux and Solaris. The tool obtained complete platform independence. And the main focus was on regression testing of the integration functional tests. This includes various protocols testing using proprietary protocol simulator as main tool that triggers application logic under test. Although most of the tests were already executed at least once, it became difficult to run regressions, as with end milestone approaching number of test cases began to grow (speaking about few hundreds of the test cases dealing with various scenarios and protocols – CAP [4], TCP [5], SIP [6], LDAP [7], Diameter [8], SOAP [9], SMPP [10], SMTP [11], POP3 [12]) and more important it was rather problematic to check all the logs for errors. When various servers, against which tests were run, were introduced, situation got even more complicated because of their different configuration they had. Not to mention error-prone process because of large number of small actions that should be repeated.

Basic procedure was the same for all test cases – create configuration, start tracing on the platform, run test script, stop tracing on the platform, check script traces, and check platform traces. It was important not to omit generation of report at the end with statistics which could take great amount of time and effort because it is needed to update test cases list, mark those which have failed, make some notes why they failed and for few hundred of test cases – it can take a while.

First idea was to write just a simple shell script that would execute all the tests and analyze the results from log files – but after a while (when it is realized that tests will be required to run with different configurations against different

 ISBN: 978-1-61208-165-6

servers) it is realized that could be benefited from real test framework.

Keyword-driven testing, which enables executing of the test scripts at a higher level of abstraction, was considered to be used as a framework. The idea of keyword driven testing is similar to that of a service or subroutine in programming where the same code may be executed with different values [13], what would make it a perfect choice for the required automation.

III. WHY ROBOT FRAMEWORK

After careful analysis Robot Framework [14] was found to satisfy all needed requirements. It is created in Python which can be implemented on all major platforms. Therefore, multiplatform requirement was completely fulfilled. Among other open source tools, Robot Framework seems to be one of the very few tools, which supports multi platform environment and it is maintained regularly, as it is listed on [15]. The tool is sponsored by Nokia Siemens Networks and released under Apache 2.0 license, meaning it is allowed to be used for free (quite important topic, not only these days).

Robot Framework is a generic, application and technology independent framework. It has a highly modular architecture illustrated in the Figure 1.

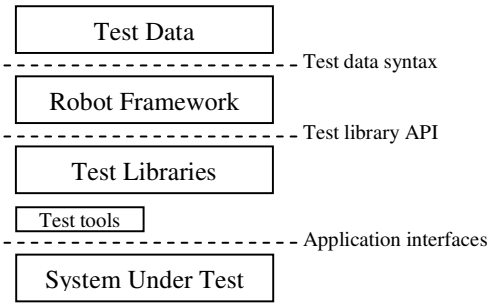

Figure 1. High level architecture [14]

The test data is in simple, easy-to-edit tabular format. When Robot Framework is started, it processes the test data, executes test cases and generates logs and reports. The core framework does not know anything about the target under test, and the interaction with it is handled by test libraries. Libraries can either use application interfaces directly or use lower level test tools as drivers [14].

What was missing was the GUI - for easy test case adding and editing. After considering options, it was decided to use RIDE, which stands for Robot Framework Integrated Development Environment [16]. Its purpose is to be an easy-to-use editor for creating and maintaining test data for Robot Framework. It is still in alpha state, but surprisingly stable for 0.3 version.

Robot Framework is a keyword-driven test automation framework [17]. Test cases are stored in HTML files (in a form of a ordinary HTML tables, as shown in TABLE I.) and make use of keywords implemented in test libraries to drive the software under test, while test suites are created from files and directories so it's convenient to store into any version of control system.

TABLE I. USING HTML FORMAT

Setting	**Value**	**Value**	**Value**
Library	OperatingSystem		
Library	lib/MyLibrary.py		

Variable	**Value**	**Value**	**Value**
${MESSAGE}	Hello, World!		

Test case	**Action**	**Argument**	**Argument**
My Test	[Documentation]	Example test	
	[Setup]	Some Setup	
	[Timeout]	5 minutes	
	Log	${MESSAGE}	
	Check If Directory Exist	/tmp	
	[Teardown]	Some Finish	
Another Test	Should Be Equal	${MESSAGE}	Hello, World!

Keyword	**Action**	**Argument**	**Argument**
Check If Directory Exist	[Arguments]	${path}	
	Directory Should Exist	${path}	

It is possible to create new higher-level keywords by combining and grouping existing keywords together. These keywords are called user keywords to differentiate them from lowest level library keywords that are implemented in test libraries. The syntax for creating user keywords is very close to the syntax for creating test cases, which makes it easy to learn - TABLE I. Rules that should be followed is that keyword names should be descriptive, clean and they should explain what the keyword does, not how it does it.

IV. REAL LIFE EXAMPLE

A. *Test suite creation*

One way to mitigate mistakes, which arise when new tool usage is started, is to create scripts that will provide immediate pay back [1]. That is, create scripts that won't take too much time to create yet will obviously save manual testing effort and, more important, by creating the scripts you will learn more about the tool's functionality and learn to design even better scripts. Not much is lost if these scripts are thrown away since some value has already been gained from them. Since Robot Framework is based on keywords, and combination of keyword can form a new user keyword - it can be seen as a script.

Robot Framework has some libraries already defined (for example, OperatingSystem, Telnet, String, Collection, etc.), but since it is Python based tool, it is easy to extend it with

 ISBN: 978-1-61208-165-6

libraries written in Python or Java. What is needed is just to write your own function and return some value (if needed).

```
def FTP_Delete(self, host, user, pwd,
file_remote):
  ftp = ftplib.FTP()
  ftp.connect(host, 21)
    try:
      try:
        ftp.login(user, pwd)
        ftp.delete(file_remote);
        return True
      finally:
        ftp.quit()
    except:
      traceback.print_exc()
      return False
```

Figure 2. New library keyword (FTP Delete) definition in Python

Writing and including own library with newly defined keywords it is easy – example for deleting file on FTP server is shown in Figure 2. When using newly defined keywords in the Robot Framework it is only necessary to replace "_" with spaces and new keyword is ready for usage.

RIDE has keyword completion feature that shows the keywords that are found either from the test suite, resource being edited, from its imported resource files or libraries. Also arguments are validated automatically for all known keywords and validation is shown on the grid editor and visualized as different cell backgrounds (everything ok – white background, too many or too few arguments - red background, optional argument - light gray, and if no arguments are allowed then cell background is dark gray). This feature works for built-in and user defined keywords.

Descriptive keywords are one of the Robot Framework features, and with RIDE possibility to create keywords, it is possible to describe test case first and then to actually create keywords and fill them with actions.

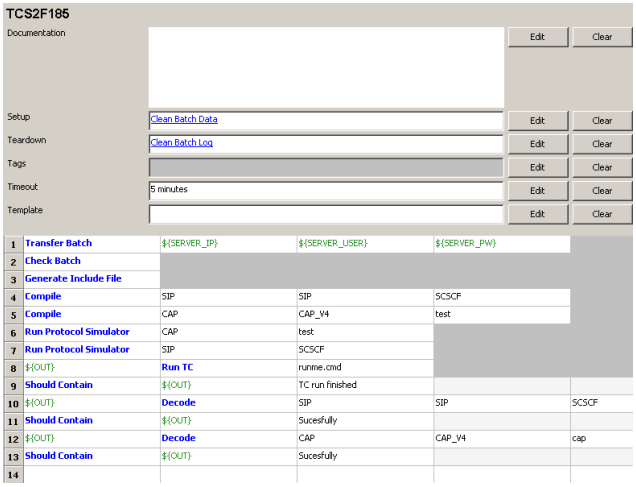

Figure 3. Test case definition in RIDE

Other thing that can happen is to find out that some sequence is needed to be used repeatedly. In that case it is possible to group that sequence, and define it as new keyword. It is easy task in RIDE - it is just needed to mark the sequence and RIDE will extract those lines and create the new keyword with auto recognition if parameters are needed. After new keyword creation RIDE will replace the sequence and change the test case accordingly.

Keywords and variable definition can be saved into resource file, so it can be used in various suites. It is a good idea if the keyword could be useful also to other tests to move it to shared resource. This way, those keywords can be used later by other tests and duplicate work is avoided.

Usually, there is a need for some setup and cleaning actions – this is also supported and, not only on the test case level, but setup and teardown actions can also be defined on the suite level.

TABLE II. TEST CASE DEFINITION IN HTML FORMAT

Test case	**Action**	**Argument**	**Argument**
TCS2F185	[Setup]	Clean Batch Data	
	[Timeout]	5 minutes	
	Transfer Batch	${SERVER_IP}	${SERVER_USER}
	Check Batch		
	Generate Include File		
	Compile	CAP	test
	Compile	SIP	SCSF
	Run Protocol Simulator	CAP	Test
	Run Protocol Simulator	SIP	SCSF
	${OUT}	Run TC	runme.cmd
	Should Contain	${OUT}	TC run finished
	${OUT}	Decode	SIP
	…	CAP	
	Should Contain	${OUT}	Call finished sucesfully
	[Teardown]	Clean Batch Log	

All this helps to read test cases, even for non technical persons, since we used live language grammar and our test case have execution defined as "Transfer Batch", "Check Batch", "Generate Include File", "Compile", "Run Protocol Simulator", "Decode Output", "Should Contain *something*" as shown in Figure 3. and in native HTML format in TABLE II.

B. Test case execution

It is possible to execute suite or just some test cases directly from the RIDE GUI, however there is a need to run test cases from the command line so its execution could be easily automated – for example from some continuous integration server. Since Robot Framework is command line tool this is usually done this way. That way various switches

can be used. All possible switches are shown and explained with running tool with "—help" switch. One of many things that can be specified (via test case name pattern matching) is the critical test cases definition. In order to complete the test suite successfully, all critical test cases have to pass.

After executing our test suite HTML report is generated, as shown on Figure 4. and the background color undoubtedly tells whether the whole test suite finished correctly. Critical test cases must be specified with a caution. If critical test cases pass successfully, regardless of other test cases results, the report will be marked as OK. However, statistics will show the number of test cases failed and specify these cases, if any.

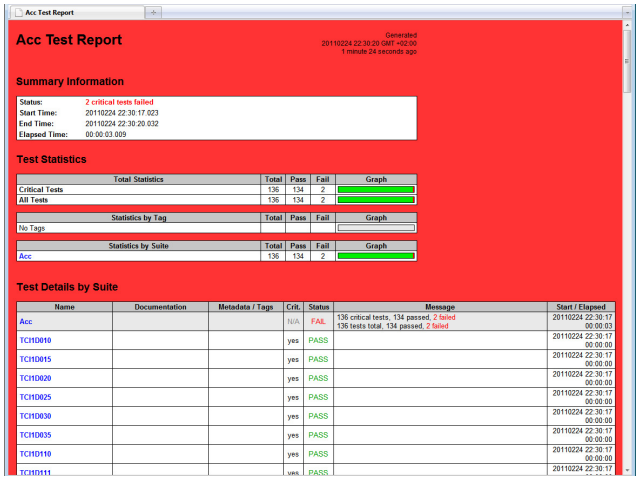

Figure 4. Test case report file

For further manual analysis, there is also detailed log file generated, as shown on I (also configurable with command line switch) with all actions, detailed description of the input and output parameters and keyword output with marked actions that went wrong. There is a keyword "Log" defined, so it is also possible to write additionally whatever need to the log file.

Since all output, as input also, is in the HTML format and already nicely formatted – it is very convenient to use it for reporting.

Robot Framework also generates XML output file which can be used for further analysis. In the source distribution there are interesting tools, for example "risto.py", used for generating graphs about historical statistics of test executions and "robotdiff.py" tool for generating diff reports from multiple Robot Framework output files.

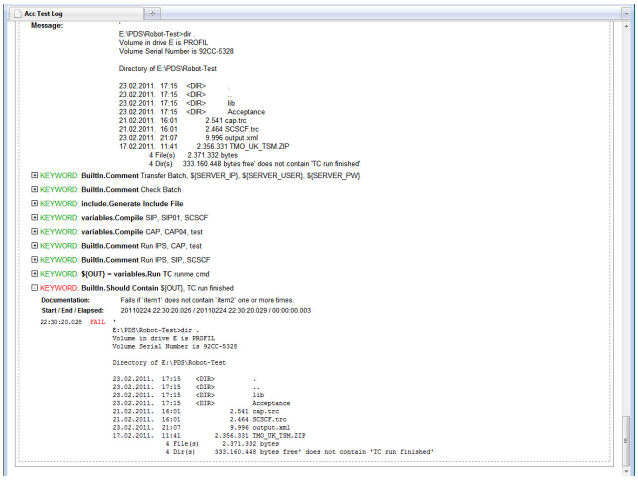

Figure 5. Test case log file

V. BENEFITS OF THE AUTOMATION

An automated test suite can explore the whole product every day. A manual testing effort will take longer to revisit everything. So, the bugs automation does find will tend to be found sooner after the incorrect change was made. Debugging is much faster, which is also meaning – cheaper, when there's only been a day's worth of changes. This raises the value of automation.

Automated tests, if written well, can be run in sequence, and the ordering can vary from day to day. This can be an inexpensive way to create something like task-driven tests from a set of feature tests.

Before Robot Framework execution of the test suite took about two days with one person executing test cases sequentially and looking for traces and, most important, being busy all that time. With Robot Framework whole process take only few hours, but only one batch command is needed to run, so person is not busy during test suite execution and can work on other topics, as shown in Table III.

TABLE III. USED TIME COMPARISON

	Time used (in hours)	
	Manual	***Automated***
Preparation of one test case	8:00	8:00
Execution of one test case	0:02	0:02
Check of one test case	0:05	0:01
Automation of one test case	-	2:00
Report for one test case	0:03	0:00
Total time used for one test case	8:10	10:03
One test run cycle	0:10	0:03

 ISBN: 978-1-61208-165-6

	Time used (in hours)	
	Manual	***Automated***
For 100 test cases - one suite run	16:40 tester involved	5:00 machine time
20 suite runs	**333:20 tester involved**	**100:00 machine time**
20 suite runs with automation time included (suites run time + automation time for all test cases)	333:20	300:00

VI. CONCLUSION

Benefit of working with Robot Framework is that writing test cases follows natural work flow with test case preconditions, action, verification and finally cleanup. Real language is used for keyword description, so it's easy to follow test case – even for non technical person, which, together with its simple usage and easy library extension, make it great tool for test case automation.

Everything is checked automatically and all reports are automatically generated and published on the web pages. This also saved lot of time when decision to introduce continuous integration was made.

The cost of automating a test is best measured by the number of manual tests prevented from running and the bugs it will therefore caused to miss [21], and this is probably the biggest strength of the Robot Framework.

REFERENCES

[1] B. Lei, X. Li, Z. Liu, C. Morisset, and V. Stolz, Robustness Testing for Software Components, Science of Computer Programming, Volume 75 Issue 10, 2010, pp. 879-897

[2] R. Patton, Software Testing, Sams Publishing, 2005

[3] K. Zallar, Practical Experience in Automated Testing, METHODS & TOOLS, Global knowledge source for software development professionals, Volume 8, Spring 2000, pp. 5-9

[4] 3GPP, Customised Applications for Mobile network Enhanced Logic (CAMEL) Phase 4; CAMEL Application Part (CAP) specification (Release 6), TS 29.078 6.3.0, September 2004

[5] RFC: 793, TRANSMISSION CONTROL PROTOCOL DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION, Information Sciences Institute University of Southern California, September 1981

[6] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, Network Working Group, Request for Comments: 3261, June 2002

[7] M. Wahl, T. Howes, and S. Kille, Network Working Group Request for Comments: 2251, Lightweight Directory Access Protocol (v3), December 1997

[8] H. Hakala, L. Mattila, J-P. Koskinen, M. Stura, and J. Loughney, Network Working Group Request for Comments: 4006, August 2005

[9] E. O'Tuathail and M. Rose, Network Working Group Request for Comments: 3288, Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP), June 2002

[10] SMPP Developers Forum, Short Message Peer to Peer Protocol Specification v3.4 Issue 1.2, October 1999

[11] J.B. Postel, RFC 821 - SIMPLE MAIL TRANSFER PROTOCOL, Information Sciences Institute University of Southern California, August 1982

[12] M. Rose, Network Working Group Request for Comments: 1460, Post Office Protocol - Version 3, June 1993

[13] A.M. Jonassen Hass, Guide to Advanced Software Testing, ARTECH HOUSE INC, 2008

[14] http://code.google.com/p/robotframework/, May 2011

[15] http:// www.opensourcetesting.org/, May 2011

[16] http://code.google.com/p/robotframework-ride/, May 2011

[17] P. Laukkanen, Data-Driven and Keyword-Driven Test Automation Frameworks, Master Thesis, HELSINKI UNIVERSITY OF TECHNOLOGY, February 2006

[18] R.W.Rice, Surviving the top ten challenges of software test automation, In Proceedings of the Software Testing, Analysis & Review Conference (STAR) East 2003. Software Quality Engineering, 2003.

[19] W.E.Lewis, Software Testing and Continuous Quality Improvement, AUERBACH PUBLICATIONS, 2005

[20] J.Bach, Test Automation Snake Oil, Windows Technical Journal, pp. 40–44, October 1996.

[21] B.Marick, When Should a Test Be Automated? Proc. 11th Int'l Software/Internet Quality Week, May 1998.

 ISBN: 978-1-61208-165-6