

Joni Ollanketo

RF PERFORMANCE TESTING USING ROBOT FRAMEWORK

RF PERFORMANCE TESTING USING ROBOT FRAMEWORK

Joni Ollanketo
Bachelor's thesis
Fall 2016
Information Technology
Oulu University of Applied Sciences

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, Langattomat laitteet

Tekijä: Joni Ollanketo

Opinnäytetyön nimi: RF performance testing using Robot Framework

Työn ohjaajat: Antti Seppälä, Pete Pietilä, Kari Jyrkkä

Työn valmistumislukukausi ja -vuosi: Syksy 2016

Sivumäärä: 35 + 1 liite

Opinnäytetyön tilaajana toimi Oy LM Ericsson Ab:n Oulun yksikkö.

Työn tavoitteena oli toteuttaa vanhan TTCN-3-pohjaisen testijärjestelmän tilalle uusi testijärjestelmä käyttäen Python-ohjelmointikieltä sekä Robot Frameworkiä.

Testijärjestelmän käyttötarkoituksena on mitata RF-suorituskykyä piensolutukiaseman tuotekehityksessä.

Käytössä olevalle TTCN-3-pohjaiselle järjestelmälle ei ollut saatavilla enää käyttötukea, joten sen korvaamiselle etsittiin vaihtoehtoja.

Pythonista ja Robot Frameworkista oli aiempaa kokemusta yrityksessä erityyppisessä testauksessa, jolloin myös työn jatkokehitys olisi helpompaa. Robot Framework mahdollistaa myös testijärjestelmän paremman integroinnin CI-järjestelmiin, jos se nähdään tarpeelliseksi.

Järjestelmän perusominaisuudet saatiin onnistuneesti toteutettua aikarajan sisällä sisältäen tarpeelliset Robot Framework-kirjastot, dokumentoinnin, käynnistyskriptit sekä lähdekoodin paketoinnin asennuspaketiksi.

Asiasanat: Python, Robot Framework, Ericsson

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Wireless devices

Author: Joni Ollanketo

Title of thesis: RF performance testing using Robot Framework

Supervisors: Antti Seppälä, Pete Pietilä, Kari Jyrkkä

Term and year of completion: Fall 2016

Pages: 35 + 1 appendix

This thesis was commissioned by Oy LM Ericsson Ab R&D site in Oulu.

The objective of this thesis was to develop a proof-of-concept of a new testing system to replace the previous TTCN-3-based system using Python and Robot Framework.

The test system was used for measuring and validating the RF performance in a small cell base station development.

The motivation behind replacing the used TTCN-3 system was that the TTCN-3 platform did not have the product support anymore.

Robot Framework and Python are used in other types of testing inside the company thus finding developers for maintaining and developing new system would be easier.

Robot Framework allows also a better integration with continuous integration tools if needed.

The basic functionality of the system was successfully implemented before the deadline, including the needed Robot Framework libraries, documentation, execute-scripts and packaging of the code for an easy installation and deployment.

Keywords: Python, Robot Framework, Ericsson

PREFACE

This thesis was commissioned by Oy LM Ericsson Ab and the work was conducted during summer 2016 in Oulu.

Big thanks to my supervisors Pete Pietilä and Antti Seppälä who provided me the subject for this thesis. Also thanks to Veijo Ruikka who provided technical guidance during my thesis work.

I also want to thank the whole Ericsson's Oulu team for all the opportunities, support and guidance I have received during my studies. They have made it possible for me to develop professionally from early on in a great work environment.

Oulu, 15.10.2016

Joni Ollanketo

CONTENTS

TIIVISTELMÄ	3
ABSTRACT	4
PREFACE	5
CONTENTS	6
VOCABULARY	7
1 INTRODUCTION	9
2 RF CONFORMANCE TESTS	10
2.1 TS 137.141	10
2.1.1 Base Station maximum output power	11
3 TTCN-3	13
4 ROBOT FRAMEWORK	15
4.1 Keywords	16
4.2 Test data	17
4.3 Libraries	17
4.4 Support tools	18
4.5 Test case styles	19
4.5.1 Keyword-driven style	19
4.5.2 Data-driven style	19
4.5.3 Behavior-driven style	19
5 SYSTEM ARCHITECTURE	21
5.1 Devices	24
5.1.1 PyVISA	25
5.2 Logging	28
5.3 Reporting	28
5.4 Installation	30
5.5 Documentation	30
5.6 Executing	31
5.6.1 Resource Files	31
6 CONCLUSIONS	32
REFERENCES	33
APPENDICES	35

VOCABULARY

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
BW	Bandwidth
CLI	Command-Line Interpreter
DUT	Device Under Test
E-ARFCN	EUTRA Absolute radio-frequency channel number
ETSI	European Telecommunications Standards Institute
FCC	Federal Communications Commission
GPIB	General Purpose Interface Bus
HTML	HyperText Markup Language
JVM	Java Virtual Machine
LTE	Long Term Evolution
MTC	Main Test Component
PTC	Parallel Test Component
reST	reStructuredText
RBS	Radio Base Station
RF	Radio Frequency

RX	Receiver
SUT	System Under Test
TSV	Tab-separated Values
TTCN-3	Testing and Test Control Notation version 3
TX	Transmitter
UARFCN	UMTS Absolute radio-frequency channel number
VISA	Virtual Instrument Software Architecture
WCDMA	Wideband Code Division Multiple Access
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
YAML	YAML Ain't Markup Language

1 INTRODUCTION

Conformance testing is used to verify that a product meets the minimum requirements set for it in a standard or a specification (1).

During the development cycle of a product, it usually goes through various different types of testing. One fundamental difference between usual testing and conformance testing is that the conformance tests criteria for passing the tests must be specified in the standard or specification.

Conformance testing aims to ensure the compatibility and interoperability between different systems.

This thesis work was commissioned by Ericsson's Oulu R&D site. The site was established in 2012 mainly to focus on the research and development of a small cell base station (2).

The object of this thesis was to develop an alternative system as a proof-of-concept for replacing previous TTCN-3 based RF performance testing system. This test system is used for testing RF conformance in a small cell base station development.

The TTCN-3 based system did not have product support anymore and due to its exotic programming language, finding new developers for maintaining it in the future might be challenging.

Robot Framework and Python were considered for replacing the previous system because of Python's popularity and experience using Robot Framework in different type of testing inside the company.

2 RF CONFORMANCE TESTS

RF conformance tests aim to ensure that all RF parts of the SUT work according to the specifications and standards.

One of the organizations which create these standards is 3GPP (3).

The requirements for multi-standard radio base station RF conformance testing are specified in the ETSI TS 137.141 document. This document provides specifications for the RF conformance testing of base stations which supports multiple radio standards, e.g. LTE and WCDMA. (4.)

2.1 TS 137.141

The document includes general test condition information, requirements and RF conformance test cases, which are divided into transmitter and receiver tests. (4.)

Some of these test cases and configurations have also regional specifications, which might provide additional limits to test cases.

Specified test cases follow a standard format which contains the following parts:

- Definition and applicability
- Minimum requirement
- Test purpose
- Method of test
- Test requirements

The document has examples of measurement system setups for different test cases.

One of the simplest setups is a transmitter setup for base station output power, transmitter ON/OFF power, modulation quality, transmitter spurious emissions and operating band unwanted emissions tests (figure 1).

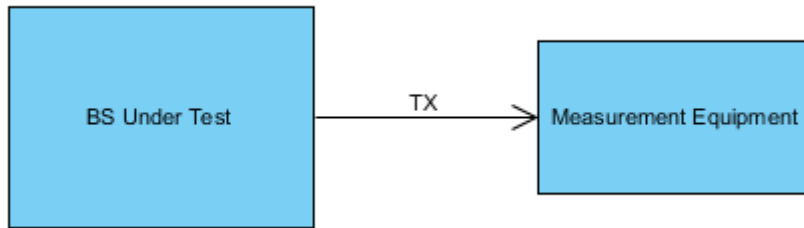


FIGURE 1. Measuring system setup for various base station transmitter tests

The test document can specify a test environment as normal or extreme.

The normal environment has defined minimum and maximum limits for a barometric pressure, temperature and relative humidity.

In the normal environment the power supply condition is defined as nominal, which means a normal operating voltage defined by the manufacturer. Vibration is specified as negligible.

2.1.1 Base Station maximum output power

This test measures the maximum output power delivered to a load with the resistance equal to a nominal load impedance of the transmitter (4).

The purpose of this test is to verify the accuracy of the maximum carrier output power in normal and extreme conditions.

The initial test condition is normal, but a test should also be run with an extreme power supply condition on one ARFCN (GSM), UARFCN (WCDMA) or E-ARFCN (LTE).

The minimum requirements for passing this test are specified in the ETSI document TS 37.104 subclause 6.2.1:

In normal conditions, the maximum carrier power should be within ± 2 dB compared to the configured carrier power declared by the manufacturer.

In extreme conditions the limit is $\pm 2,5$ dB.

These are minimum limits which do not take test tolerance value (TT) into account. This value is added to a minimum requirement.

In this test case TT is specified as follows for UTRA and E-UTRA:

$$0,7 \text{ dB} \quad f \leq 3,0 \text{ GHz}$$

$$1,0 \text{ dB} \quad 3,0 \text{ GHz} < f \leq 4,2 \text{ GHz}$$

Thus, the maximum carrier power in normal conditions should be $\pm 2,7$ dB when using the carrier frequency below 3,0 GHz and $\pm 3,0$ dB when using the carrier frequency between 3,0 GHz and 4,2 GHz.

In extreme conditions these values are $\pm 3,2$ dB and $\pm 3,5$ dB.

The basic testing procedure itself is fairly simple:

- Start the Base Station transmit using maximum power defined in the test configuration using defined test models or a set of physical channels
- Measure the mean power for each carrier

3 TTCN-3

TTCN-3 is a scripting language developed and standardized by ETSI and it is used typically for the protocol, service, module and API testing (6).

The main building block of TTCN-3 is a module, which contains a definition part and an optional control part as seen below.

```
module MyTestSuite {
    // This module contains a definitions part...
    :
    const integer MyConstant := 1;
    type record MyMessageType { ... }
    template MyMessage { ... }
    :
    function MyFunction1( ... ) { ... }
    function MyFunction2 { ... }
    :
    testcase MyTestcase1 { ... }
    testcase MyTestcase2 { ... }
    testcase MyTestcase3 { ... }
    :
    control {
        // ... and a control part
        :
        var Boolean MyVariable := true; // local variable
        :
        MyTestCase1;
        MyTestCase2;
        if (MyVariable) MyTestCase3;
        :
    } // End control
}
// End module
```

Modules can be parametrized and they can import definitions from other modules.

The control part in the module executes the test cases specified in the definition and called inside the control partition.

The parallel execution can be achieved by using parallel test components (PTC). This is one of the three different types of components (7):

- Main Test Component (MTC)
- Parallel Test Component (PTC)

- PTC that defines Abstract Test System Interface

Components communicate by using communication ports, which can have direction in, out or inout (6).

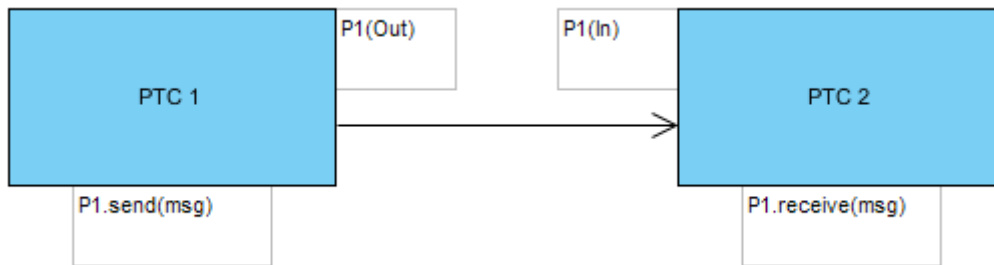


FIGURE 2. Communication between components

Ports can have three different types:

- Message-based
- Procedure-base
- Mixed

MTC is automatically created when the test execution starts, but PTCs must be explicitly created by calling create operation.

Components can be connected inside the test system by calling a connect operation:

```
connect(PTC_1:P1, mtc:P1)
```

The map operation is used for connecting components to an abstract test system interface:

```
map(PTC_1:P2, system:PCO)
```

4 ROBOT FRAMEWORK

Robot Framework is an open source keyword-driven test automation framework based on the Python programming language. It uses a keyword-driven approach to testing. Robot Framework runs on Python, Jython (JVM) and IronPython (.NET). (8.)

The high-level architecture of the Robot Framework test environment is shown in figure 3.

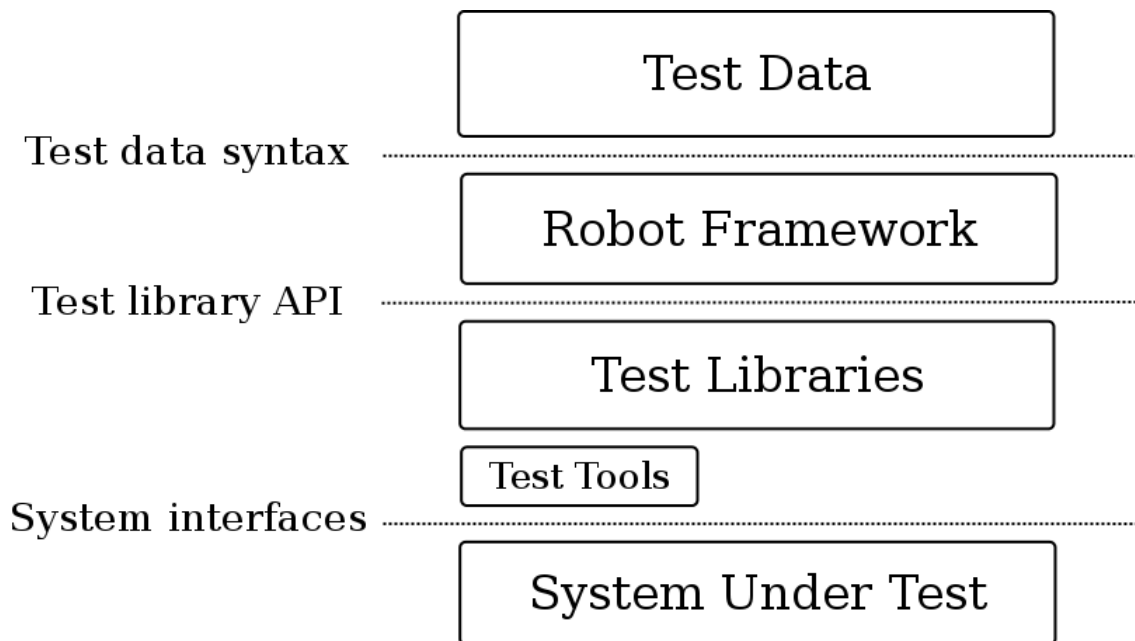


FIGURE 3. Robot Framework high-level architecture

The test case execution starts by calling Robot Framework with test data as input arguments. Robot Framework parses test suites to run from test data and starts executing them. (9.)

The controlling of the test tools and other needed subsystems inside the test suite is done through test libraries. These libraries can be used, for example, to

start a process on the test machine, to start remote connections and to parse data.

After the execution ends, report and log files are normally created in the XML and HTML format. (9.)

4.1 Keywords

The testing functionality in a test case is created using 'keywords', which is the main idea behind the Robot Framework concept (9).

Keywords can be implemented as methods inside test libraries or test data files. Library keywords provide lower level functionality to Robot Framework, thus using them directly inside the test case might impair understandability of the test case flow.

The keywords implemented in test data files can wrap multiple keyword calls as one higher level keyword (10). This can improve the test case readability and provide a higher level functionality for test cases.

```
*** Settings ***

*** Test Cases ***
Keyword Test
    Echo And Timestamp    Hello, World!

*** Keywords ***
Echo And Timestamp
    [Arguments]          ${arg}
    Log To Console       ${arg}
    ${Time} =            Get Time
    Log To Console       ${Time}
```

This is a test case, which has a keyword definition with one input argument.

4.2 Test data

Test data can contain test suites, resource and variable files (10).

Test cases can be arranged in a hierarchical structure using directories that contain test case files. Test suite directories can also contain sub-directories, which create lower level test suite and this nesting can continue as deep as needed.

Test case files provide the lowest level of the test suite hierarchy. These files can contain one or more test cases. (10).

Data files are written as tables using HTML, TSV, reST or plain text formats. The used format is recognized by a file extension. These files can contain four types of test data tables: settings, variables, test cases and keywords. (9).

The following example shows the basic structure of a test data file. It is written in a plain text format using spaces as a column delimiter. The plain text format would also allow the use of pipe character with spaces as column delimiter.

```
*** Settings ***
Library      OperatingSystem

*** Variables ***
${MSG}      Hello, world!

*** Test Cases ***
TC1
    [Documentation]    Example test
    Print Message     ${MSG}

*** Keywords ***
Print Message
    [Arguments]       ${IN_MSG}
    Log                ${IN_MSG}
```

4.3 Libraries

Libraries provide an interface to test suites for interacting with the system under test and test tools. This interacting is done through library keywords, which are called from the test case or higher level keywords.

Robot Framework includes several standard libraries for basic operations and its functionality can be extended with external third-party libraries.

Standard libraries provide a basic functionality for interacting with the operating system, modifying data, creating telnet connections, running processes, modifying XML files and more (8).

Extending the Robot Framework functionality further can be done by writing your own test libraries. These libraries can be written using Python or Java. In case of Java libraries, the running environment is limited to Jython.

4.4 Support tools

The Robot Framework package contains four different built-in support tools, which can help achieving different tasks before and after the test execution. (9).

- Rebot tool is used for post-processing output XML files after the test execution.
- Libdoc is a tool used for generating the HTML documentation of the keywords in test libraries and resource files.
- Testdoc tool is used for generating a high level documentation of the test case files.
- Tidy can be used for cleaning up test output data and changing output file formats.

4.5 Test case styles

Robot Framework test cases can be written using many different test case styles, which all have their pros and cons.

Keyword-driven and behavior-driven styles are basic test case styles that describe the test case workflow.

The data-driven style uses a test template as one keyword with inputs thus the same test can be run with multiple inputs.

4.5.1 Keyword-driven style

Keyword-driven tests use keywords to describe their execution flow. The example below shows basic keyword driven test case. This case uses keywords 'Evaluate' and 'Should Be Equal' to set and verify variable values.

```
*** Test Cases ***
Keyword-Driven Test
  [Documentation]      Keyword-Driven Test
  ${ANSWER} =         Evaluate  ${2} + ${1}
  Should Be Equal    ${ANSWER}    ${3}
```

4.5.2 Data-driven style

In the data-driven test case style, the test case is created as a template with different variables as input data.

This approach can be very useful when the test case must be tested with multiple inputs and outputs. (2)

4.5.3 Behavior-driven style

The behavior-driven style implements executable requirements, which are known from ATDD (Acceptance Test Driven Development).

These executable requirements can be written in a Given-When-Then style like in BDD (Behavior Driven Development).

```
*** Test Cases ***  
Valid Login  
  Given login page is open  
  When valid username and password are inserted  
  and credentials are submitted  
  Then welcome page should be open
```

As can be seen from the example above, this test case style is easily understandable also to persons without a technical background.

5 SYSTEM ARCHITECTURE

Since this was a major rework on a large existing system with a totally different language with its own features and qualities, the architecture had to be carefully planned.

The system architecture was designed to be easily extendable to different test systems and equipment by adding test libraries and changing the used resource files.

One requirement for the new system was to keep output report files identical compared to the existing system. This required writing of my own listener-library, which generates output XML-files according to the XSL-file, which specifies the XML format.

The TTCN-3 based test system required the Windows operating system, but this is not the case when running tests with Robot Framework since it runs also on Linux and Mac OS X.

There might be some restrictions in the device specific libraries (e.g. the device does not work in the Linux environment), but the basic VISA-device libraries and other standard functionality should be able to run on Linux, Windows and Mac OS X.

A generalized high-level picture of a typical test place can be seen in figure 4. Test devices could represent e.g. power meters, signal analyzers and signal generators.

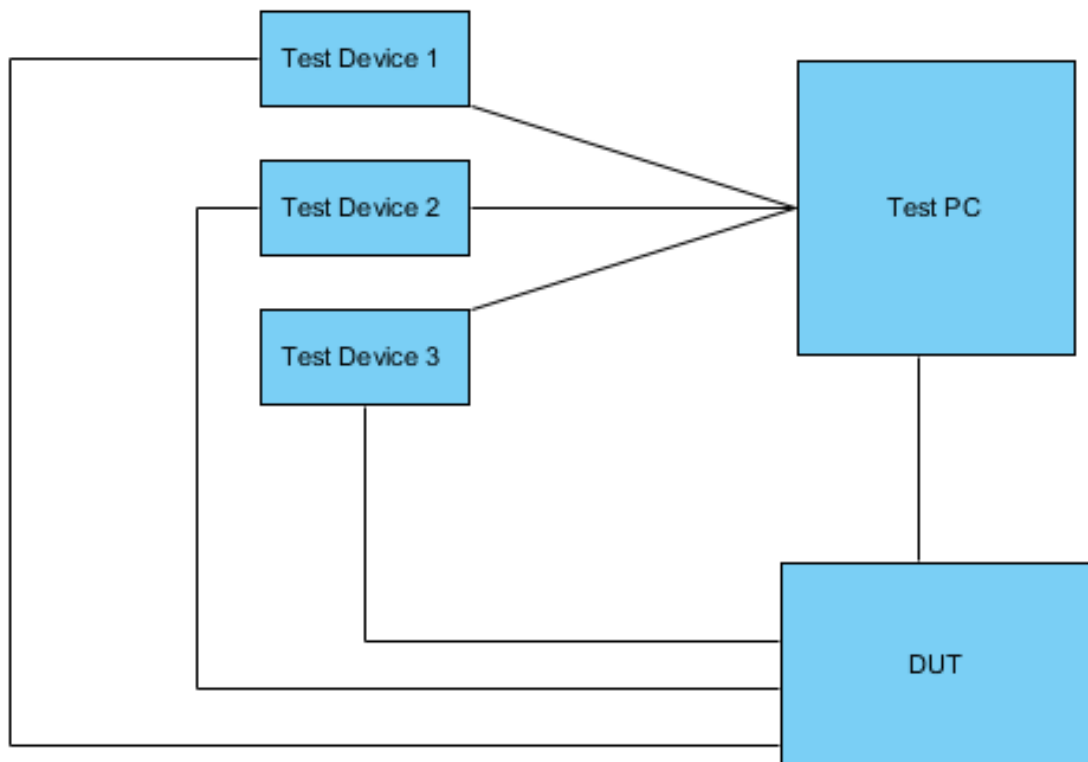


FIGURE 4. General test place.

The high-level architecture of the testing system is described in figure 5. This picture shows the main components that are needed for executing the test suites.

Most of the functionality is implemented inside test libraries, which interact with lower levels of the system.

These libraries include the controlling of the DUT, test equipment and other required devices and systems.

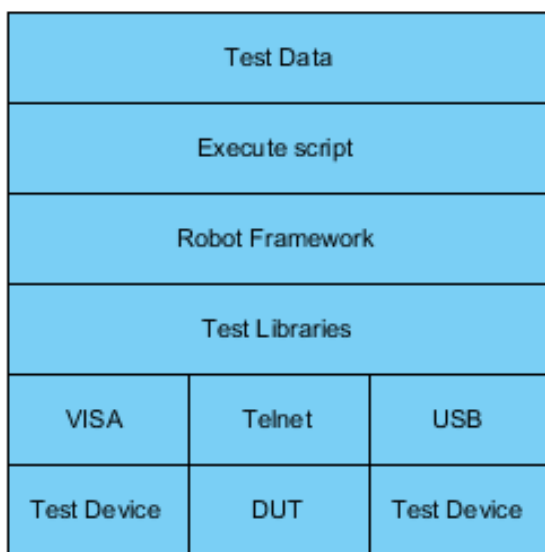


FIGURE 5. High-level architecture of the test system

Some higher-level functionality is also implemented in resource test data files by using keywords.

5.1 Devices

The RF performance testing requires the use of various test equipment, and all of these devices require test libraries for controlling them. Similar devices usually share some common methods and commands which should be implemented only in one place.

This is achieved through the use of inheritance and abstract base classes.

The abstract base class can be used to define the type of the device and what methods it must implement.

Figure 6 shows an example of an inheritance tree with three abstract base classes which define the methods and properties for a VISA device, signal analyzer and signal generator.

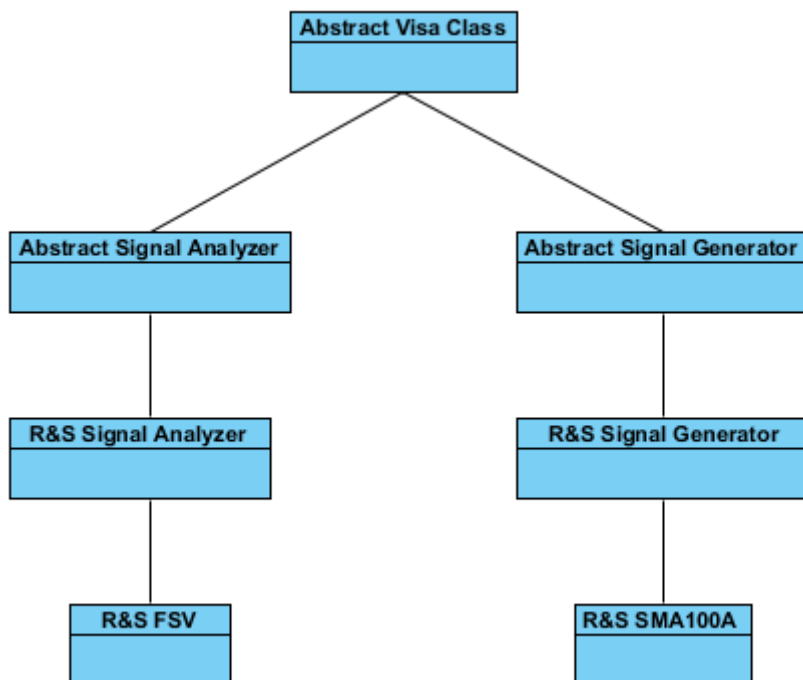


FIGURE 6. Inheritance tree example

The R&S Signal Analyzer class must implement abstract methods and properties defined in the Abstract Visa Class and Abstract Signal Analyzer class.

R&S FSV and R&S SMA100A classes are test libraries which Robot Framework can import.

Test equipment is mainly controlled through the PyVISA-library which wraps VISA controls inside Python libraries.

5.1.1 PyVISA

PyVISA is an open source Python package used for controlling devices using VISA (11).

It works on multiple operating systems and VISA-backends. It can even use a simulated PyVISA-sim backend which is used to simulate the test equipment.

This could be useful while developing new test cases and libraries since expensive test equipment does not have to be reserved for developing purposes.

PyVISA-sim simulated devices which are defined in a YAML-file (12).

Simulated devices provide basic responses to queries and can have properties with different data types and value ranges.

These properties can have setter and getter methods which can be used to request and modify the property values.

```
sense_avg_count_auto_nsratio:
  default: 0.01
  getter:
    q: 'SENS:AVER:COUN:AUTO:NSR?'
    r: '{:.2f}'
  setter:
    q: 'SENS:AVER:COUN:AUTO:NSR {:.2f}'
  specs:
    min: 0.0
    max: 1.0
    type: float
```

The example above defines one property for a simulated device. The value formatting in the setter and getter response uses the PEP 3101 formatting.

A full example of a simulated device is given below. This is a reduced version with only few dialogues and properties, but it shows the basic structure of the device.

```
spec: '1.0'
devices:
  NRPZ:
    eom:
      GPIB INSTR:
        q: '\r\n'
        r: '\n'
    error: ERROR MSG
    dialogues:
      - q: '*RST'
      - q: '*IDN?'
        r: 'ROHDE&SCHWARZ,NRP-Z23,MOCK'
      - q: '*TST?'
        r: '0'
      - q: 'INIT:IMM'
      - q: 'FETC?'
        r: '46.8325462e+000;46.8234552e+000;46.8123472e+000'

      #SYSTEM INFO
      - q: 'SYST:INFO? MANUFACTURER'
        r: 'Rohde & Schwarz GmbH & Co. KG'
      - q: 'SYST:INFO? TYPE'
        r: 'NRP-Z23'
      - q: 'SYST:INFO? STOCK NUMBER'
        r: '1137.7506.02'
      - q: 'SYST:INFO? SERIAL'
        r: '123456'
      - q: 'SYST:INFO? HWVERSION'
        r: '000000000'
      - q: 'SYST:INFO? HWVARIANT'
        r: '000000000'
      - q: 'SYST:INFO? SW BUILD'
        r: '<build number>'
      - q: 'SYST:INFO? TECHNOLOGY'
        r: '3 Path Diode'

    properties:
      sense_avg_count:
        default: 4
        getter:
          q: 'SENS:AVER:COUN?'
          r: '{d}'
        setter:
          q: 'SENS:AVER:COUN {d}'
        specs:
          min: 1
          max: 65536
          type: int

resources:
```

```
GPIB::0::INSTR:
  device: NRPZ
```

This simulates a GPIB device which has an address 0.

If the above file is saved with a name 'example.yaml', it could be tested in Python CLI using the following commands:

```
>>> import visa
>>> rm = visa.ResourceManager('example.yaml@sim')
>>> rm.list_resources()
('GPIB0::0::65535::INSTR',)
```

This shows that the resource manager is initialized successfully with the defined file using PyVISA-sim. The part after the @-character defines what backend is used.

Simulated devices can be instantiated and queries can be tested with the following commands:

```
>>> inst = rm.open_resource('GPIB0::0::INSTR', read_termination='\n')
>>> inst.query('*IDN?')
u'ROHDE&SCHWARZ,NRP-Z23,MOCK'
```

National Instruments NI-VISA is a commonly used backend for real VISA devices. Ideally, everything else would stay the same when using a real equipment except that the ResourceManager instantiation would be modified to use a real backend.

A typical module stack used for controlling a device from Robot Framework is presented in figure 7.

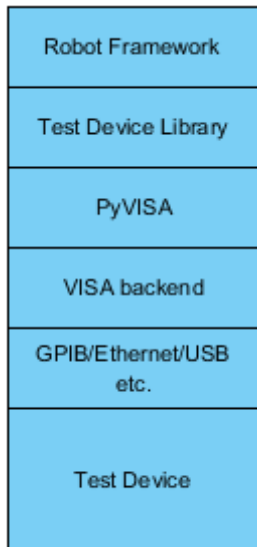


FIGURE 7. Test device control model

5.2 Logging

In addition to Robot Frameworks' internal logging, an external logging service was introduced to create a hierarchical logging for all specified libraries. This logging service uses Python's logging facility.

The logging hierarchy is defined by library types, for example all VISA devices have one common 'parent' logger.

The external logging service provides an additional control over logging of the test runs, since it uses Python logging handlers, which could e.g. send the logged data to a centralized server.

5.3 Reporting

Reporting is done by an external Robot Framework listener class in addition to existing Robot Framework reporting utilities. This class creates basic XML and HTML reports according to the XSL file in the same format as in the TTCN-3 test system.

The HTML file can be generated from the XML file by a XSL transformation using the XSLT file (Figure 8).

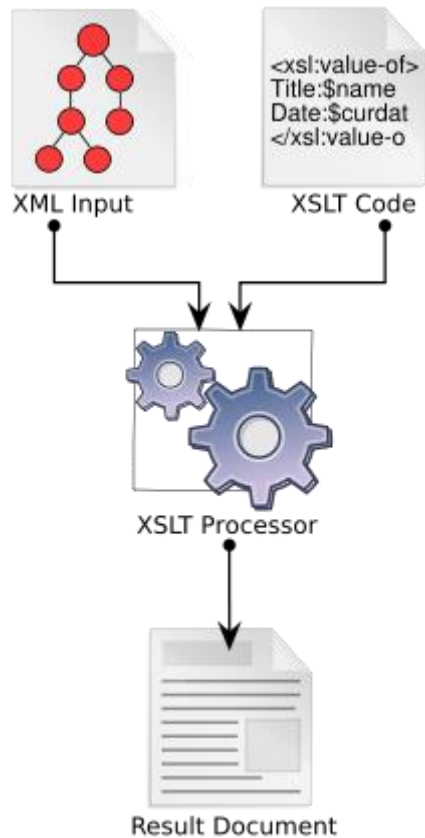


FIGURE 8. XSL Transformation

Generated reports and documents are moved to a single output directory after the test execution.

The reporting library can also append test data to the existing XML file, if the file name was specified in execute arguments. This can be useful when running multiple test cases at different times but wanting only one test report.

5.4 Installation

The whole codebase is contained within a single repository, which contains the Python install script.

The script uses Python setuptools for managing dependencies, package installation, upgrading and uninstallation.

5.5 Documentation

The basic keyword documentation is done inside test libraries using Python docstrings. The test library documentation is generated using the Robot Frameworks libdoc-tool.

The general documentation is written in the reST format and generated using the Sphinx documentation tool.

This documentation style has few upsides:

- Documentation is included in the repository
 - Easier to maintain
 - Version control
 - Easier to find
- Sphinx can generate a documentation in various output formats:
 - HTML
 - LaTeX
 - ePub
 - Texinfo
 - Manual pages (POSIX man)
 - Plain text

5.6 Executing

Test cases are started by the Python script, which parses the required parameters including test place specific information, output directory for generated logs and output files, test suite to run and other Robot Framework specific parameters.

The test case execution is started from Python script by using Robot Frameworks API.

5.6.1 Resource Files

Resource files are used to define specific variables and other information which are needed to specify test place and test suite parameters.

Test place parameters define the used test place. This definition contains test place specific parameters such as what equipment the test place contains, amongst other things.

Resource files also include parameters for test case pass limits.

6 CONCLUSIONS

The main objective of this thesis was to develop a proof-of-concept for replacing the previous TTCN-3 based RF performance test system using Robot Framework and Python.

Since my background was in the system level test automation using Java, the RF performance testing was altogether a different kind of environment. Python and Robot Framework were also mainly new to me before I started working on this thesis, but they proved out to be easy to learn on the go due to a good documentation and a simple syntax.

The proof-of-concept was successfully developed inside the time limit, which implemented the basic functionality needed from the system.

Although replacing the previous TTCN-3 based system would still require a major development work, it was proven that it would be possible using Robot Framework and Python libraries. This implementation would also provide some additional flexibility to the test system, such as simulated test equipment and the use of centralized database for test place definitions.

Overall, I believe this thesis work proved out to be a successful experiment which taught me valuable new skills and provided a solid layout for further development.

REFERENCES

1. Gray, Martha – Goldfine, Alan – Rosenthal, Lynne – Carnahan, Lisa
2010. Conformance Testing. National Institute of Standards and Technology. Available at:
<https://www.nist.gov/itl/ssd/information-systems-group/conformance-testing>. Date of retrieval 19.11.2016.
2. Klemettilä, Pasi 2012. Ericsson aikoo suunnitella Oulussa pienen tukiaseman. Kaleva. Available at:
<http://www.kaleva.fi/uutiset/oulu/ericsson-aikoo-suunnitella-oulussa-pienen-tukiaseman/567002/>. Date of retrieval 19.11.2016
3. About 3GPP Home. Available at: <http://www.3gpp.org/about-3gpp/about-3gpp>. Date of retrieval: 19.11.2016.
4. ETSI TS 37.141 V13.4.0 (2016-10) Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; E-UTRA, UTRA and GSM/EDGE; Multi-Standard Radio (MSR) Base Station (BS) conformance testing (3GPP TS 37.141 version 13.4.0 Release 13). Technical Specification 2016. Sophia Antipolis Cedex: ETSI. Available at:
http://www.etsi.org/deliver/etsi_ts/137100_137199/137141/13.04.00_60/ts_137141v130400p.pdf. Date of retrieval 20.11.2016.
5. ETSI TS 37.104 V13.3.0 (2016-10) Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; E-UTRA, UTRA and GSM/EDGE; Multi-Standard Radio (MSR) Base Station (BS) radio transmission and reception (3GPP TS 37.104 version 13.3.0 Release 13). Technical Specification 2016. Sophia Antipolis Cedex: ETSI. Available at:
http://www.etsi.org/deliver/etsi_ts/137100_137199/137104/13.03.00_60/ts_137104v130300p.pdf. Date of retrieval 20.11.2016.

6. ETSI ES 201 873-1 V4.8.1 (2016-07) Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI Standard 2016. Sophia Antipolis Cedex: ETSI. Available at: http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.08.01_60/es_20187301v040801p.pdf. Date of retrieval: 21.11.2016.
7. Grabowski, Jens – Ulrich, Andreas 2004. An Introduction to TTCN-3. Available at: <http://www.ttcn-3.org/ttcn-3uc04/cd/Programme/Education-Track/20030426-TTCN-3-Introduction.pdf>. Date of retrieval 19.11.2016
8. Robot Framework. Available at: <http://robotframework.org/>. Date of retrieval 20.11.2016.
9. Robot Framework User Guide. 2015. Nokia Solutions and Networks. Available at: <http://robotframework.org/robotframework/3.0/RobotFrameworkUserGuide.html>. Date of retrieval 20.11.2016.
10. RobotFrameworkAdvancedGuide. 2011. Available at: <https://twiki.cern.ch/twiki/bin/view/EMI/RobotFrameworkAdvancedGuide>. Date of retrieval: 21.11.2016.
11. PyVISA: Control your instruments with Python. 2015. Available at: <https://pyvisa.readthedocs.io/en/1.8/>. Date of retrieval 21.11.2016.
12. PyVISA-sim: Simulator backend for PyVISA. 2016. Available at: <https://pyvisa-sim.readthedocs.io/en/latest/>. Date of retrieval 21.11.2016.

APPENDICES

Appendix 1 Memo of initial data (in Finnish)

LÄHTÖTIETOMUISTIO

Työn tiedot	Tekijä ¹ Joni Ollanketo	Tilaaaja ² Oy LM Ericsson AB
	Tilaaajan yhdyshenkilö ja yhteystiedot ³ Pete Pietilä Antti Seppälä	
	Työn nimi ⁴ OpenTTCN-pohjaisen testiympäristön korvaaminen Robot Frameworkillä	
	Työn kuvaus ⁵ OpenTTCN:llä kehitetyt testit halutaan siirtää toiseen testiympäristöön, koska OpenTTCN-työkalulle ei ole tukea jatkossa. Kehitetyt testit liittyvät pääasiassa tukiaseman RF-suorituskyvyn conformance-testaukseen.	
	Työn tavoitteet ⁶ -Kehittää tarvittavat apuluokat Robot Frameworkillä ja Pythonilla tarvittavien testilaitteiden ja ympäristöjen käyttöön -Portata osa olemassa olevista testeistä uuteen ympäristöön -Tuottaa koodista selkeä dokumentointi muille testaajille ja automaatiokehittäjille	
	Tavoiteaikataulu ⁷ 1.5.2016 - 31.8.2016	
	Päiväys ja allekirjoitukset ⁸ / / Tekijän allekirjoitus	
<ol style="list-style-type: none"> 1. Tekijän nimi, puhelinnumero ja sähköpostiosoite. 2. Työn teettävän yrityksen virallinen nimi. 3. Sen henkilön nimi ja yhteystiedot, joka yrityksessä valvoo työn suoritusta. 4. Työn nimi voi olla tässä vaiheessa työnimi, jota myöhemmin tarkennetaan. 5. Työ kuvataan lyhyesti. Siinä esitetään muun muassa työn tausta, lähtötilanne ja työssä ratkaistavat ongelmat. 6. Esitetään lyhyesti ja selvästi työn tavoitteet. 7. Esitetään projektin tavoiteaikataulu. Silloin, kun työllä on välitavoitteita, myös ne merkitään aikatauluun. Tavoiteaikataulun ja oppilaitoksen yleisaikataulun perusteella tekijä laatii oman aikataulunsa. 8. Lähtötietomuistio päivätään ja sen allekirjoittavat tekijä ja tilaaajan yhdyshenkilö. 		