UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Essi Passoja

EVALUATION OF TWO PARALLEL BASE STATION TESTING SYSTEMS

Master's Thesis
Degree Programme in Computer Science and Engineering
March 2024

Passoja E. (2024) Evaluation of Two Parallel Base Station Testing Systems. University of Oulu, Degree Programme in Computer Science and Engineering, 49 p.

ABSTRACT

This paper describes two testing systems; Flame and a test automation system using Robot Framework, both of which have been developed within Nokia Solutions and Networks Oy. During the writing of this paper I was working in a Base Transceiver Station software test automation team. In test automation, all tests are run by automation and the tester's responsibility is to make sure the automation is working and analyzing the test cases when failures occur, report issues as well as planning new test cases. This diploma has been made as a part of the test automation system change -project, where the old system Flame is replaced using an open-source framework Robot Framework. The project planning to switch to a new test automation system using RF was started in the fall of 2021, and Flame will be officially dropped by the end of May 2024.

After the new Robot system had been in use for a year, the usability of each system was evaluated using a questionnaire. The questionnaire revealed further improvement and training needs, but also the benefits in comparison to Flame. Flame still had an advantage over Robot from the usability point of view, but the differences were not radical and will likely improve over time as the Robot system develops further and the system becomes more familiar to all its users.

Keywords: Test Automation, Robot Framework, base transceiver station, BTS testing, testing systems

Passoja E. (2024) Kahden tukiasematestiohjelmiston arviointi. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 49 s.

TIIVISTELMÄ

Tämä diplomityö kuvaa kahta eri testiautomaatiosysteemiä; Flamea ja Robot Frameworkiä, jotka ovat molemmat kehitetty Nokia Solutions and Networks Oy:ssä. Tätä työtä kirjoittaessani työskentelin tukiaseman softan testaukseen keskittyvässä tiimissä. Tiimimme hyödyntää testiautomaatiota isona osana testausta, jolloin testaajan vastuulle jää vain varmistaa että testiautomaatio toimii ja vikastatuksen saaneiden testiajojen analysointi ja raportoiminen, sekä uusien testien suunnittelu. Tämä työ on kirjoitettu osana projektia, jossa vaihdetaan nykyinen testisysteemi Flame uuteen systeemiin, joka kirjoitetaan pelkästään Robot Frameworkillä. Testiautomaatiosysteemin vaihdosprojektin RF:iin pohjautuvaan systeemiin aloitettiin syksyllä 2021 ja Flamen käyttö lopetetaan virallisesti Toukokuun 2024 lopussa.

Kun uusi Robot systeemi oli ollut käytössä vuoden ajan, sen käytettävyyttä arvioitiin kyselyn avulla. Kyselyn tulokset paljastivat jatkokehitys - ja koulutuskohteita, mutta myös etuja Flameen nähden. Flame oli kyselyn perusteella Robotin edellä sen käytettävyydessä, mutta erot näiden systeemien välillä eivät ole radikaaleja ja todennäköisesti kaventuvat ajan myötä, kun Robot systeemi kehittyy pidemmälle ja systeemi tulee tutummaksi kaikille sen käyttäjille.

Keywords: Testautomaatio, Robot Framework, tukiasema, tukiasematestaus, testisysteemit

TABLE OF CONTENTS

FOREWORD

This master’s thesis was done while working for Nokia Solutions and Networks Oy during 2021-2024.

Thank you to my supervisor Aku Visuri, for supporting me during this long process and providing invaluable feedback throughout the project.

Thank you to my supervisor Pekka Lonnakko from Nokia Solutions and Networks, for allowing me to do this project from the beginning by providing assistance and feedback whenever I would need any. Thank you also to my other amazing colleagues for all your help and support, and my line manager for allowing me to take on the project and write this thesis based on it.

Thank you also to my partner and my family, for pushing me forward and helping me in any way you could to support me during my studies.

Oulu, March 12th, 2024

Essi Passoja

LIST OF ABBREVIATIONS AND SYMBOLS

ALD	Antenna Line Device
API	Application Programming Interface
BTS	Base Transceiver Station/Base Station
CM	Cloud Manager
FHS	Front-Haul Switch
FM	Fault Management
GUI	Graphical User Interface
HW	Hardware
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IoT	Internet of Things
KPI	Key Performance Indicator
KW	Keyword
MHA	Masthead Amplifier
MN	Mobile Networks
OM KPI	Operability & Maintenance Key Performance Indicator
PoV	Point of View
PM	Performance Management
RET	Remote Electrical Tilt
RF	Robot Framework
SSH	Secure Shell Protocol
SUT	System Under Test
SW	Software
TA	Test Automation
TAF	Test Automation Framework
UE	User Equipment
UTE	Unified Test Environment
VM	Virtual Machine

1. INTRODUCTION

Testing software (SW) and hardware (HW) is an important phase when creating new products or developing old ones, as it ensures the quality of the product as it gets in the hands of a customer. There are different testing phases as well as various ways to test the software. Some examples of different test types are integration testing, unit testing, functional testing, regression testing and performance testing. All of these testing types have different objectives, but the idea behind them is always the same; to find flaws or improvement needs in the software [1].

Testing can be performed both manually and automatically. In manual testing, the tester performs all test phases without using any automation tools [2]. Manual testing is often and ideally done before the test can be automated, and sometimes it is necessary for a longer period of time if the automation is not yet capable of performing the test due to the system under test not supporting automatization or if the system is working so unpredictably that it is more sensible to keep on doing manual tests until the software becomes more stable. During manual testing, the phases of the automated test can be defined and also made sure that the test is suitable for automation.

Automated testing is time efficient and requires less human resources. This allows the tester to allocate more time to making detailed analysis in fault investigation, developing their skills and knowledge, and creating new tests to provide a wider test scale. Automation provides an effortless way to run tests simultaneously in multiple test environments. Tests can be triggered to be started at a given time or when a new software is released for testing. Scheduling helps save electricity, which has become an issue especially in the European Union - the automation can be triggered at times when the electricity is cheaper and the usage is at its lowest [3].

This thesis focuses on testing base transceiver stations (BTS). A BTS is a building block of a mobile network and it needs to be tested SW and HW wise. The test automation (TA) developed in this thesis is made for testing BTS SW. This team has been focused on running automated tests for a long time. The existing TA framework is called Flame and it offers a graphical user interface (GUI). It will be changed to a new framework that will be used company wide, a Robot TA system based on Robot Framework. The automation systems are compared based on usability.

- RQ1: "How does the ease-of-use differ between each automation system?"
- RQ2: "How efficient it is to use each system when creating new tests or making changes to existing ones?"

The research questions are evaluated based on the answers from a questionnaire, that is deployed within the team. In addition to the research questions, the questionnaire also covers the previous programming experience of the users, as well as the programming skill development during the process.

Ease-of-use is estimated from two view points; how easy the users experience the test creation or editing, and how easy they experience to read and understand the logs that are provided by each TA system.

The efficiency of each system is very close to the ease-of-use, and it is measured using the estimated times that the users report for test creation and test edits.

The questionnaire result evaluation revealed that the Flame is experienced to be more efficient and easy to use, however, the test edit times also suggested that there may be a need for further training on how to use the Robot system more efficiently. The usability over all is expected to improve over time for Robot, as the users gain more experience of the new system.

2. BACKGROUND

This thesis was done at Nokia Solutions and Networks Oy (later referred to as Nokia), an international company offering their customers, who are mostly mobile operators, hardware and software for building and maintaining mobile networks. Nokia offers 2G, 3G, 4G and 5G products (including the physical system and the software that runs in it) commercially, often using dynamic spectrum sharing in order to offer smooth transition from older networking technologies to newer ones. This way newer and older generations of products can be used together in a non-standalone system [4].

Nokia is divided to four different sectors; Mobile Networks (MN), IP and Fixed Networks, Cloud and Network Services and Nokia Technologies [5]. All teams within business groups have their own dedicated purposes. In MN, such purposes include, for example, developing new features or products, testing, developing and maintaining tools for being used within other teams.

On top of the commercialised products, Nokia is also developing 6G products in its Bell Labs, which are planned to be commercialised in 2030 [6]. With 5G and 6G being developed for the Internet of Things (IoT) purposes, the importance of the reliability of our networking systems become greater by the day. It is essential to test different scenarios to ensure the equipment will function as specified, without causing wider down times in the network or having unacceptably slow data transfer times.

This thesis work is done in a team within Mobile Networks -business group, where the focus is on functional testing of software in a Base Transceiver Station (BTS), or a base station in short. The team is divided to five main areas within the test scope; Performance Management (PM) -counters, Antenna Line Devices (ALD), Fault Management (FM) & Recovery, Operability & Maintenance Key Performance Indicators (OM KPI) and Startup. The testing areas will be discussed further in Chapter 3.2.

The team is focused on automation testing, which offers a more efficient and thorough way of testing. In the team, testers and automation developers are separated, which allows testers to focus on keeping up with the specifications of the software and the tested products while taking care of the testing and problem reporting, and designing new tests. This leaves the automation developers more time to focus on fixing old scripts and developing new ones.

2.1. Base Transceiver Station

To create a wireless communication grid, base stations are needed. A BTS consists of system modules, baseband modules, radio modules and cells. A simple BTS configuration is shown in Figure 1. All of these modules have different functions and the tests are observing the status of each unit to verify correct functionality. In general, the system module is where the BTS software is running and it is controlling the system behaviour, a baseband module holds the capacity for data/calls, while radio and antennas act as transceivers and receivers of the data.

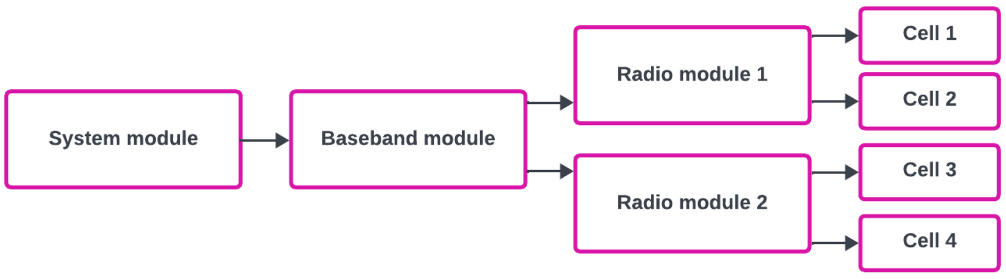


Figure 1. A simple base station configuration with a system module and a baseband module, 2 radio modules, each having 2 cells.

In a customer site the base station configuration can be much more complicated in comparison to a simple configuration in Figure 1, but for testing purposes it is often sufficient to perform tests on a simple configuration. Nevertheless, it is essential not to exclude more complex configurations from the test scope, as they often uncover distinct issues. Complexity in a BTS can be caused by, for example, including a front-haul switch (FHS) attached to a baseband module, as shown in Figure 3, allowing multiple radios to be attached to one baseband module port, or chaining radios, as shown in Figure 2, meaning that a radio is attached to another radio, also resulting in multiple radios attached to one baseband module port. In our team, a lot of testing is done on environments with a simple BTS. However, due to the diversity of complex customer site configurations, there also exists a need for similar test environments in testing, resulting in a need for the TA system to be able to adapt also to the more complex test environments.

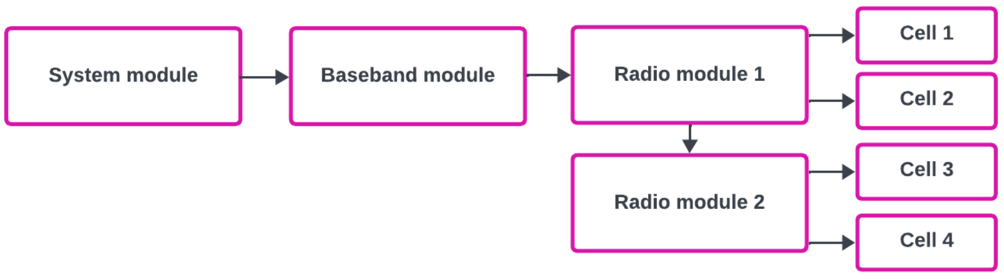


Figure 2. A base station configuration where 2 radios are in chain so that only 1 baseband module port is in use.

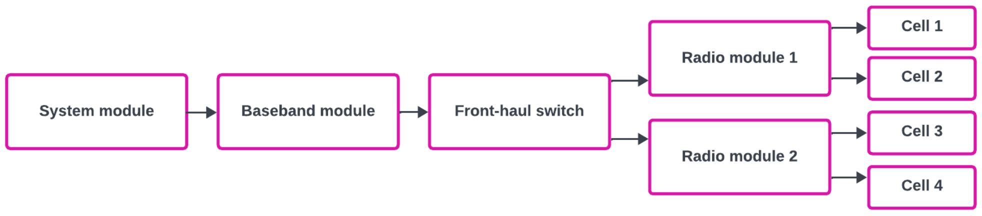


Figure 3. A base station configuration where 2 radios are connected to a front-haul switch so that only 1 baseband module port is in use.

To accommodate this diverse array of test environments effectively, a dynamic automation system enables correctly analyzed BTS behaviour during a test run. Without this dynamic TA approach, there is a need to prepare a separate script for each test environment to validate the correctness of test results, leading to inefficiencies when a lot of time is used to create files for different configurations.

2.1.1. Test Environment

The tested BTS and any connected components are referred to as the test environment. Typically, the environment includes either a virtual machine (VM) or a physical computer, serving as an interface for the BTS. A VM or a computer offer a possibility to view the graphical user interface of the BTS. The GUI offers different functionalities, such as locking and resetting different units. All operations present in the BTS GUI, including configuration management and system controls, can be executed via an Application Programming Interface (API), which facilitates automation and thus extends the testing scope to all functions available in the GUI.

The computer linked to the BTS can also be used to form a Secure Shell Protocol (SSH) -connection to the BTS and specific units. Via the SSH connection, it is possible to send commands directly to units, allowing for a wide range of testing scenarios. For instance, it can be used to simulating faults, such as causing the radio to report a malfunction or terminating processes within the BTS system module. These commands are exclusively used for testing purposes and differ from the GUI functionalities, which can also be performed by the customer on their own sites. Although these functionalities may not be directly utilized by customers, they play a vital role in testing. They enable the creation of controlled situations to evaluate specific system behaviours that can occur randomly in customer configurations. Such test scenarios are typically within the scope of FM and recovery testing, ensuring the BTS behaviour is occurring according to specifications.

In addition to the computer attached to the BTS, various other components like switches and specialized log collectors can be connected to either the BTS or the computer. Switches have different functionalities, that can implement, for example, detaching and reattaching wires. This functionality proves invaluable in testing scenarios, as it helps verify whether the BTS effectively communicates accurate fault notifications to users and, more broadly, ensures that the system's functionality adheres to the specified requirements.

2.1.2. A Dedicated Test Environment

A dedicated test environment is static, as its configuration remains unaltered unless unit changes are required. A single user (the tester) has full control over its setup and the scope of testing it accommodates. Yet, this level of control comes hand in hand with the tester's responsibility to address unit failures and replacements.

When a tester uses a dedicated environment, they are required to do the TA installations themselves. Robot Framework and the required libraries are mostly installed using pip. Pip is a tool that can be used for installing, updating and uninstalling libraries that are not a part of Python standard library[7].

Both scripting tools, whether Robot Framework or Flame, can be executed via the Jenkins automation tool, which offers a web-based GUI. Jenkins is used in local environments to automate scheduled testing in order to remove the need to use commands on command line if the tester wishes to do so[8]. The GUI is especially handy when Robot Framework is used, as it does not have its own GUI, unlike Flame.

2.1.3. Unified Testing Environment - UTE

When a tester has their own test environments, it was often noticed that the usage ratio of the environment is low and the tests are consistently run on the same configurations, which may affect which issues are found during testing. To address both of these challenges, the concept of Unified Test Environment (UTE), also known as UTE Cloud, was established. UTE enables the reservation of test environments, while also handling test scheduling, for example for the regression testing (Regression testing is further explained in Chapter 2.2.1).

UTE is developed and maintained by dedicated teams. These teams are responsible for tasks such as managing maintenance requests, setting up new environments, and decommissioning outdated ones or environments with no usage, as well as the development of the scheduler automation and the automation that prepares the environment for testing, i.e. the Cloud Manager (CM). During environment preparations, CM handles, for example, downloading and installing the libraries and applications that the tester has defined, prepares the BTS configuration to be as expected and sets the test environment into the wanted state.

A UTE team is also responsible for fixing any hardware related issues. Using varied hardware in software testing sometimes creates challenges, for instance, if a radio malfunctions and its impact surfaces only sporadically—perhaps once a week or even a month—it becomes challenging to conclude whether the problem resides in the hardware or the BTS software. Addressing this concern, the UTE team collects statistics from test results to identify consistent failures tied to specific hardware. However, instances where issues appear solely during certain test scenarios can prolong the process of isolating problematic hardware. In the context of local environments, the ability to detect hardware issues is often faster, as the consistent daily usage by the same tester, under identical configurations, fosters prompt issue recognition.

By centralizing these responsibilities, testing teams are liberated from maintaining test environments, allowing them more focus toward testing. This allows the teams to use a broader spectrum of configurations in testing, thus improving the likelihood of finding bugs and issues in the software, contributing to an overall improvement in software quality.

2.2. Test Automation

Test automation is a mechanism for conducting tests with a reduction to manual labour. It optimizes resource allocation, affording human testers the opportunity to redirect their efforts in the non-repetitive and more intellectually demanding tasks, such as issue analysis and test design. With automation, test computers run tests independently without any human effort, allowing multiple tests to run simultaneously on different machines, which is a great improvement compared to manually driven tests as one tester can run only one test at a time. Test automation runs over nights, weekends and holidays without a tester being present, allowing more testing hours per piece of hardware without adding any human labour hours. It also provides the opportunity to run long tests, even for multiple days, without constant supervision.

From a cost perspective, it is important to prioritize automation whenever feasible. Automation not only accelerates processes but also mitigates the reliance on expensive human resources in comparison to manual testing. It empowers organizations to run their entire test scope every day. Running the full test scope daily or in case of new software is ready for testing, is known as regression testing.

2.2.1. Regression Testing

In terms of manual testing, regression testing requires substantial human resources and reduces the test scope due to resource restrictions, whereas automation grants the flexibility to design a larger test scope with significantly fewer human resources. Automated regression testing leads to rapid issue identification, while minimizing the amount of changes tested within a single test cycle [9][10].

Consequently, this approach improves software debugging and resolution times, allowing more time to be used in the software development. It ensures that software bugs are frequently discovered soon after the SW change is implemented, improving the product quality. Moreover, the comprehensive test coverage afforded by automated regression testing increases the likelihood of uncovering rare and infrequently occurring issues before they surface at a customer site, further solidifying the robustness and reliability of the software.

2.2.2. Functional Testing

Functional testing is also known as black box testing, meaning that the tests are written independent from the SW itself and the tests are designed based on the system specification [11].

Functional testing is an integral part of SW testing in ensuring that the system is operating according to the specifications [11]. When the testing is focused on the internal workings of the software and on the SW structure, we talk about white box -testing, but this is often done earlier on in the SW development.

The most typical functional testing tools are scripts written in Perl, TSL, JAVA script etc. Choosing the right tool to fit the project requirements can be a challenging process [11]. One framework that is easily extendable, multi-platform compatible and an open-source frameworks is Robot Framework [9], and for these reasons it has been chosen to be the main automation framework within Nokia BTS SW testing.

2.3. Test Automation Requirements

In BTS testing, gaining a comprehensive understanding of the BTS configuration and its impact on the overall system behavior during testing is very important. This knowledge is the key to effectively evaluating the system's performance. Testers use the system specifications when designing the test expectations and analysis, which is then used as the instructions to automation development.

When tests are automated, the test automation framework assumes the role of executing the tests and providing an initial assessment of the BTS system's behavior and performance. The automated analysis is not as detailed or as in depth to make a fault report without any additional investigation, but it provides a good starting point. This is significantly reducing the tester's workload, particularly when reviewing only the test cases flagged as failures by the automation.

For instance, the test automation framework can promptly alert the tester to the presence of additional or missing fault notifications. However, it typically does not offer an extensive analysis explaining the underlying reasons for these occurrences. Thus, it falls upon the tester to examine the logs collected by the automation, allowing for a more profound understanding of the issue's origin. Using this analysis, the tester can then proceed to report the fault to the software developers, thereby facilitating a more efficient debugging and resolution process.

2.3.1. Keyword Driven Testing

When testing a BTS, the scripts can quickly become long and complex. When writing the test case, keyword driven testing makes the test case script easy to read as the complex code is hidden underneath the keywords [9]. Robot Framework uses keyword driven testing. The keywords are written in Python or Robot Framework itself by combining existing keywords. Since it is an open-source project, it is possible to make the required extensions for the tests yourself. The transformation from a Python function or a method to a RF keyword is simple: a function/method 'do_something()' would simply be 'Do Something' as a keyword. Keywords (KW) and their variables are separated with two or more spaces.

Robot Framework is a keyword driven framework, which makes it very easy to read even for those without experience in coding. In this project, the usability is in focus, because most of the testers do not have any experience in writing code and they are used to a graphical user interface (GUI). Robot Framework was chosen to be the framework used throughout the company's automation due to the above reasons, but also to make it easier for employees to switch teams within the company and for more experienced teams to provide assistance to the less experienced ones without the need of familiarizing themselves to different TA systems.

3. FLAME AND ROBOT SYSTEM COMPARISON

3.1. System Overview

The Base Transceiver Station test environments, both dedicated and the ones in Unified Test Environment pool, are located in the company laboratories, within a controlled network. It is more common for the testers to use UTE environments in order to save test equipment resources, as they are used by multiple users instead of one and are therefore used more efficiently, as discussed in Chapter 2.1.3.

UTE and dedicated test environment PC's or Virtual Machines can be connected to from any location. In the context of a dedicated environment, direct access to the PC within the laboratory is also feasible. Figure 4 shows the very basic connection between the user, PC and the BTS, when the user is in the laboratory using the PC directly.

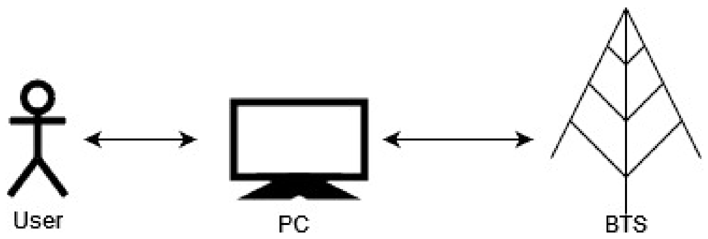


Figure 4. User can see and control the BTS using a PC or a VM connected to the BTS.

Upon accessing the PC attached to the base station, testers can run automated tests using a test automation system. The old TA system is called Flame and the new system is a TA system based on Robot Framework, called Robot later in the thesis. Both systems are currently usable and available on the test PCs (Figure 5). The main difference between Flame and Robot is that Flame boasts a graphical user interface, while Robot operates through command-line execution.

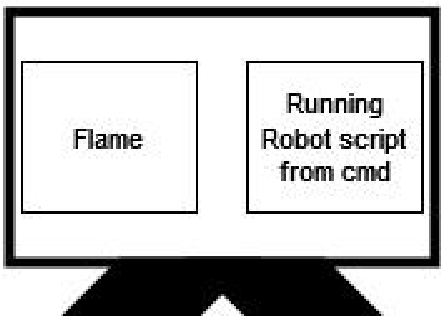


Figure 5. Robot Framework and Flame are both installed to test PCs and UTE virtual PCs. Flame tests can be run via GUI, Robot is run from command line.

3.1.1. Automation in a UTE Environment

A UTE environment operates with a high degree of automation, wherein the Cloud Manager (CM) has the integral role of importing all essential libraries during the initial phases of configuring the test environment. This is done because the environments are wiped clean when the environment is released after a previous test run to ensure it does not affect the new test.

The use of UTE environments frees the tester from the responsibility of up-keeping the test environment while providing variance to the used configurations, increasing the likelihood of finding new bugs in the base station software. The UTE Cloud system is described in Figure 6. A UTE environment is reserved using a UTE web GUI. The CM handles the reservations and the environment preparations. The tester defines the BTS configuration in the reservation form, so the CM can find and reserve the suitable one for the testing purposes.

The configuration is given to the UTE reservation system by using a topology string with labels in it. The labels include the information of the used system module, baseband modules, radios and the number of optical cables per radio. The cables indicate how many cells each radio has connected to them and how they are connected.

For test planning, Flame offers an intuitive process. After downloading test scripts onto the system, testers organize suitable test cases within the test plan through a straightforward drag-and-drop interface. The GUI's user-friendliness is evident, as it is easy to see and edit the test execution order.

In contrast, the Robot Framework demands testers to articulate the specific steps constituting the test case. Recognizing that detailed test steps may not always be pre-known, we decided to provide exemplary test cases that can be readily employed as templates, necessitating minimal to no modification for immediate deployment.

Most UTE environments are dynamic, meaning that some units are attached/detached to/from the BTS according to the given prerequisites, and the units can be used by multiple BTS's based on their availability. Generally, the CM attempts to secure a test environment within the same physical site as the user. This ensures a latency-optimized connection similar to the low latency observed in local test environments.

Once a UTE environment is reserved for the user, they can connect to the VM attached to the BTS. There the user can view the system details using the BTS GUI and perform tests. This approach requires some manual work and is mostly used when debugging TA scripts or while doing special tests either manually or with automation. A more automated option to run test cases is to use a single run -option in the UTE reservation system, in which the user gives the test script location and environment specifications for the CM, which then runs the test fully automatically and collects the logs to the UTE log database, from which they can be easily viewed and analyzed. The regression testing is launched by the CM without tester assistance, based on the information that is saved to another database.

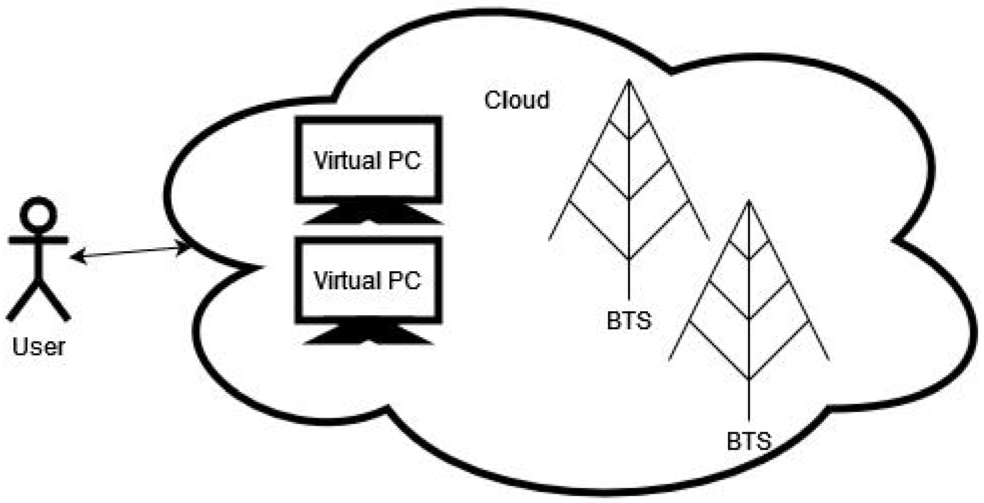


Figure 6. UTE resources offer dynamic BTS configurations. A tester can define the needed configuration for the tests and (automated) UTE manager then provides such configuration by building the base station from available resources, or attempts to find an existing and available setup.

Having a lot of variance in the test environment configurations is better for software quality, but it poses a challenge to the TA. The scripts need to be dynamic; there cannot be any hard-coded system behaviour expectations. This leads to the requirement that the TA system needs to be able to recognize the test environment configuration before starting the test.

3.2. Testing Areas

In this project the test automation is built for five different areas:

1. **Recovery and Fault Management (FM)**: This testing area focuses on breaking the BTS in various ways to test that it is able to recover by itself and/or give out the correct fault messages to the user.

2. **PM counters**: Performance management counters. The counters are used by the customer to observe the BTS functionality. In our testing, the counters are not directly observed, but instead the report file is checked to ensure it has the correct values with the correct format. This is tested in various test scenarios.

3. **ALD**: Tests Antenna Line Devices such as Masthead Amplifiers (MHA) and Remote Electrical Tilts (RET) and their effect on the BTS.

4. **OM KPI**: Operability & Maintenance Key Performance Indicators. Focus is in observing the startup times in different reset scenarios.

5. **Start-up**: Focuses on observing the start-up success in different scenarios.

In a test case the pre- and post-conditions vary, but generally it is required that the BTS is configured and is in a working condition in the beginning and in the end of the test. For example in Recovery and FM -test area, a test case starts with the precondition checks to ensure that all radios and cells are online and working, and there are no additional fault notifications active in the BTS.

In case there is a possibility to perform a data transfer test in the test setup, it is done as well. A data transfer test is performed with user equipment (UE), which is usually a phone. In a UE test environment, each UE is inside a box to ensure they attach to a specific cell, allowing the automation to recognize which cells in the BTS are working.

After the checks and a possible data transfer test, a fault is generated to the BTS and the test script is supposed to follow the behaviour of the BTS to make sure everything goes according to the specifications. During the fault generation, it is important to make sure that the correct fault notifications are appearing and nothing unexpected breaks when the fault is generated. In the end of the test, the post-conditions are checked, and the data transfer test is performed if possible.

Currently the script does not perform a deeper analysis of occurring issues, but rather the failing cases are reported to a reporting tool, where the responsible tester can then find links to logs and make a deeper analysis and fault reports to SW developers, if needed. However, the aim is to create an event log that shows the tester as precisely as possible where and when the fault has occurred, thus limiting the amount of time required to finish an analysis.

3.3. Test Case Automation

Usually a test case is run with the following steps to ensure the correct pre- and post-conditions are met, as well as ensuring that the logs contain all information throughout the testing process.

1. **Establish the BTS connection**
2. **Start log collection**
   - Collect BTS configuration
   - Start listeners
3. **Check that pre-conditions are met**
4. **Run test case**
5. **Check that post-conditions are met**
6. **Stop log collection**
   - Stop listeners
7. **Close the BTS connection**

3.3.1. Establish the BTS Connection

When the automation starts up, the first thing to do is to form a connection to the BTS API. Via the API, the automation can collect information from the BTS, its current states and fetch faults, as well as control the BTS functions.

3.3.2. Starting Log Collection

Without logs, it is impossible to see what went wrong in a test step, therefore the log collection is started as early as possible, immediately after the BTS connection is formed. The automation system's own logs are started before the connection has been created, but the event log that is created by the test script is started at this point with the other logs.

The basic logs contain syslogs, IMS logs and a Snapshot. Syslogs include the messages sent by the BTS and they are often big text files. IMS log is a recording of the activities in the BTS, and Snapshot is a collection of the system (in this case BTS) states at a certain point in time. For viewing IMS and Snapshot there is a Nokia specific tool which provides a graphical user interface (GUI). The GUI helps analyzing issues tremendously, as the tester can easily view all events taking place at one point in time in the BTS with one glance.

Collecting BTS configuration

The old TA system, Flame, collects a configuration identifier for each unit. As was discussed in Chapter 2.1, a radio has the most units attached to it, therefore it has the most amount of information in its identifier; It contains the information to which system module(s) and baseband module(s) it is attached to, as well as if the radio is chained to another radio or attached via front-haul switch.

As an example, if a radio has a name 'RMOD' and it is attached to the second system module, the third port of the first baseband module and is the second radio in chain, the configuration identifier would look like this: RMOD(2.1.3.2). This is valuable information to be collected, as it tells the tester lots of information within one identifier, and the automation is able to make conclusions when the connections between different units are known.

Since the configuration identifiers had been found very useful by the testers when using Flame, it was also brought to the Robot TA system. The algorithm was translated from C# to Python, and while doing the translation, the algorithm was updated since it included parts that were intended for old units that are no longer in the test scope.

For the TA system to understand the relations between different units, the TA needs to collect its own information, too. While running the algorithm for collecting the configuration identifiers, we form a class instance for each unit and save the configuration identifier among other useful information from the BTS configuration file to identify the units.

An API facilitated by the BTS's graphical application enables the TA system to read the BTS configuration files, removing the need for a tester to provide the BTS configuration. This way, the logs will show the tester which unit is affected, while the system itself can make conclusions what units should be affected. For example, in a situation where a baseband module was to lose power, and a radio is attached only to that baseband module, there should be a chain reaction visible where the attached radio modules and cells are lost, in case they are not attached to any other baseband or radio modules with power.

When the test environment involves special equipment, the information for them is not found from the configuration files. For instance, in scenarios involving power off/on tests, which use a remotely controllable switch, the automation system then needs the information for the switch's unique IP address and the corresponding port-unit mapping from elsewhere to ensure accurate control of the switch and correct expectations for the BTS behaviour.

To provide all the necessary information of these specialized equipment attached to the test environment, UTE uses a Python initialization -file that contains the information that can then be fetched by the TA. In the old TA system, Flame, UTE testing requires an additional environment file where basic configuration and special equipment information is given. This is a file that needs to be kept up-to-date by the tester.

When the design process for the Robot Framework TA system started, the goal was to get rid of the separate environment file. Initially the Robot file included parameters for these variables when using a local test environment, but eventually we decided to adopt the same Python initialization file as in UTE. This removed the need to create multiple Robot -files for each environment, while simultaneously making the TA very dynamic.

Starting listeners

A test case can be run with or without listeners, that are observing certain behaviours of the BTS at a constant rate. In this specific project, they are often necessary in catching some fault notifications that may appear only for a very short period time. For this reason, there is a fault notification listener that is polling the BTS every 2 seconds.

For a listener to work as expected, it is started on a separate thread in order for it to be able to act separately from the rest of the test case. Overall, the test includes 3 different threads. One is polling fault notifications, one syslog prints and the last one is for the BTS unit states.

The listeners print any changes to the event log at the time they appear, which makes it easy for the tester to understand the chronological order of events easily just by looking at the event log. If the checks were done only at specific times, it could be more challenging to see when exactly something happens, making the log analysis a slower process.

3.3.3. Checking Pre-Conditions

After the configuration is known, there needs to be checks that all units are in an expected state and there are no unexpected, active fault notifications in the BTS. Commonly the states are supposed to be in an Online state with nothing blocked or locked. These situations may arise when there are issues present in the BTS from the previous test run or there is a hardware failure.

The consequence of having unexpected states would end in unexpected results in the actual test, which makes the test result useless. For this reason, the test is interrupted at this stage to prevent wasteful use of the testing resources.

If the test includes a data transfer test, the UEs are used to test if data is being transferred between the UE and an external server. Tests with UEs are good for catching hidden issues with cells, which may not be noticed otherwise.

3.3.4. Running a Test Script

In the actual test run, many different scenarios can be performed. One example test case is to kill an application via SSH connection. In a correct situation, the BTS SW will react to the lost application by raising a fault notification and making appropriate recovery actions to start up the application again.

Tests vary largely between the testing areas, from measuring start-up times after different kinds of resets for different units, checking the success of a configuration change, and making sure the BTS is able to recover and warn the user correctly in fault situations.

If the BTS is unable to function as expected, the TA should mark the test as failed and provide some insight to the problem(s) occurring during the test. After each test step, it is important to make sure that all units and cells have certain states and the expected fault notifications are present without any disallowed fault notifications amongst them.

With syslog listeners, it is also possible to observe if there are certain prints found in the syslogs, which can imply something being wrong in the BTS. They are also a way to confirm that the automation has successfully created the issue to the BTS, as the syslog should show, for example, that an application has become unresponsive.

3.3.5. Checking Post-Conditions

In this project, the BTS status in the end of the test is almost always fault free, with everything functioning normally. To guarantee this is the case, the automation system is again checking for active fault notifications, BTS unit states, etc.

Before these checks are made, it might be worthwhile to create a wait time between the test and the post-condition checks, as some issues only become visible after a longer time period. For the sake of testing efficiency, it is not reasonable to create waiting times for over 1 hour, but even a 10-minute extra wait might catch important issues in the BTS behaviour.

3.3.6. Stopping Log Collection and Listeners

Once the test has completed and all checks are performed, the log collection is stopped in order to save the logs in their specified location. If this step fails, most logs are missing, making the test more difficult to impossible to analyze. Most commonly this happens due to script errors, however, whenever they occur, the goal is to prevent these crashes in the future by error handling, ensuring successful log collection. Over time, these failures become more uncommon for each automation system as more error situations are handled correctly.

3.3.7. Closing the BTS Connection

The BTS connection(s) are important to close successfully, as they create nodes to the BTS. Each listener thread also has their own node, therefore one test run has multiple active nodes. The amount of active nodes is limited to 9, after which the BTS refuses further connections until the existing ones are teared down.

3.3.8. Special Testing

When a tester has written a fault report, the software developers often request to perform testing with specific changes to the test or using special equipment during the test, for example, using a separate log collector from a specific unit. This usually requires only minor changes to the test itself, in order to run the test with the automation system instead of performing the test manually. Depending on the automation system, the effort required by the tester to prepare the files varies.

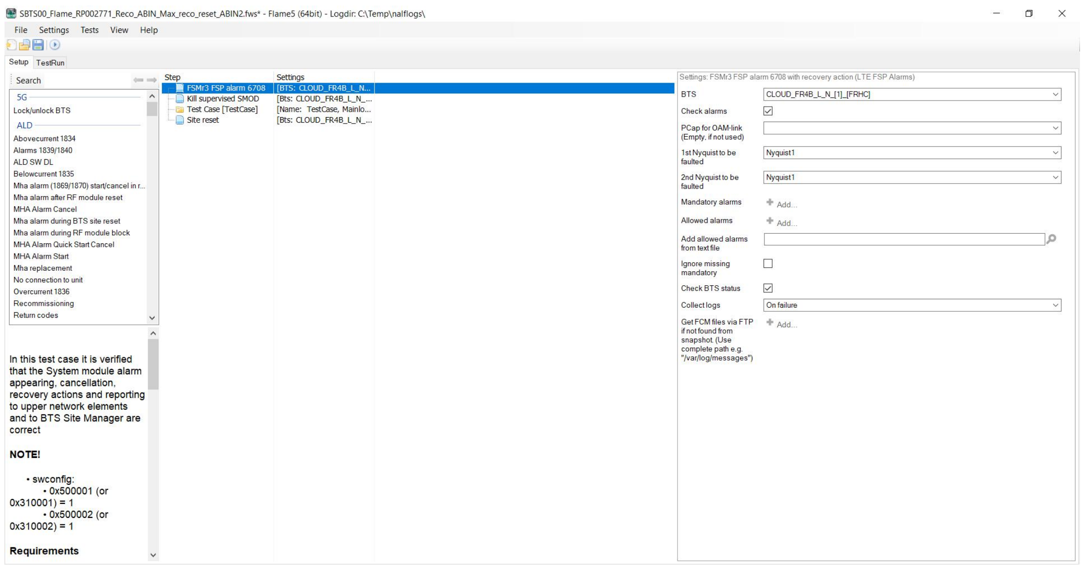
3.4. Test Automation Frameworks

In test automation, a TA framework is a set of components that make reporting and running the tests possible. Some frameworks offer a graphical user interface to make the usage of test automation easier. A GUI offers the tester an intuitive process to the test automation usage, removing the need to understand the underlying code or how to run the tests from the command line.

In this team, Flame has been in use for over ten years. As an attempt to unify the tools used within the company and thus reduce costs in the application development, there is an initiative to move away from Flame and use Robot Framework, and alongside of it, Test Automation Framework, i.e. TAF, a library of keywords for Nokia's own testing purposes.

From a business point of view (PoV), using the same tools in all teams is offering lower costs in terms of licensing and development costs, and it makes job rotation and knowledge sharing between teams easier. From a technical PoV it is more challenging to compare different tools, as there are other aspects that should be considered, than the measurable ones. Usability is an important thing to look at, but hard to measure, as it depends on the user and their skill set.

3.4.1. Flame


SBTS00_Flame_RP002771_Reco_ABIN_Max_reco_reset_ABIN2.fws* - Flame5 (64bit) - Logdir: C:\Temp\nalflogs\

5G
- Lock/unlock BTS

ALD
- Abovecurrent 1834
- Alarms 1839/1840
- ALD SW DL
- Belowcurrent 1835
- Mha alarm (1869/1870) start/cancel in r...
- Mha alarm after RF module reset
- MHA Alarm Cancel
- Mha alarm during BTS site reset
- Mha alarm during RF module block
- MHA Alarm Quick Start Cancel
- MHA Alarm Start
- Mha replacement
- No connection to unit
- Overcurrent 1836
- Recommissioning
- Return codes

In this test case it is verified that the System module alarm appearing, cancellation, recovery actions and reporting to upper network elements and to BTS Site Manager are correct

NOTE!

- swconfig:
  - 0x500001 (or 0x310001) = 1
  - 0x500002 (or 0x310002) = 1

Requirements

Step	Settings
FSMr3 FSP alarm 6708	[BTS: CLOUD_FR4B_L_N...
Kill supervised SMOD	[Bts: CLOUD_FR4B_L_N_...
Test Case [TestCase]	[Name: TestCase, Mainlo...
Site reset	[Bts: CLOUD_FR4B_L_N_...

Settings: FSMr3 FSP alarm 6708 with recovery action (LTE FSP Alarms)

BTS	CLOUD_FR4B_L_N_[1]_[FRHC]
Check alarms	✓
PCap for OAM-link (Empty, if not used)	
1st Nyquist to be faulted	Nyquist1
2nd Nyquist to be faulted	Nyquist1
Mandatory alarms	Add...
Allowed alarms	Add...
Add allowed alarms from text file	
Ignore missing mandatory	☐
Check BTS status	✓
Collect logs	On failure
Get FCM files via FTP if not found from snapshot. (Use complete path e.g. "/var/log/messages")	Add...

Figure 7. Screenshot of Flame GUI.

Flame was developed at Nokia for Nokia to offer a graphical interface for BTS TA. Flame is currently maintained by one person, making it a risk from business point of view, as the maintenance of the tool may be disrupted in case that person decides to leave the project. The holiday season can also be problematic as there are often no skilled substitutes available.

Flame offers a GUI for the tester, where the test and test environment settings can be easily defined without having to read or edit the code. This is a major benefit especially when testers are not familiar with coding, which is the case for most testers within the team, where this change is implemented in.

Sometimes setting up the test environment settings can be time consuming, if there are lots of additional tools in use, for example, power breakers to turn power off or on in specific modules from a remote computer.

When implementing new settings into a test script, the existing test file needs to be opened and saved for the new setting to be applied. This is a time consuming process and avoided whenever possible in the code, which often leads to poorer code quality.

Flame is taking advantage of its capabilities of creating listeners for the BTS unit states and the BTS logs to catch unexpected occurrences during the test. In case such an event occurs, it is printed to an event log, which makes it easy for the tester to analyse the situation when the order of events are easy to see.

Flame listeners use very little resources, as the listener is receiving information from the BTS only when there is information to be sent, removing the need for the automation system to poll the BTS regularly. This information can be processed by the automation system whenever needed.

Special testing requirements

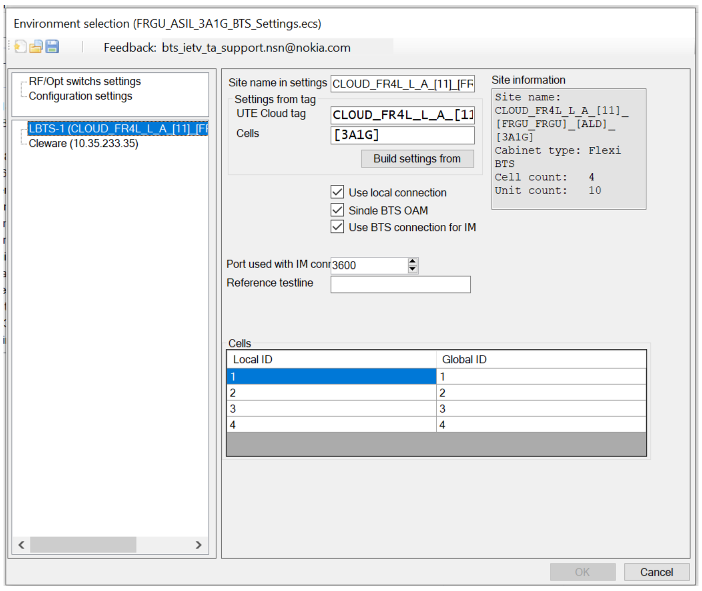
In order to perform temporary changes in a Flame test, the test file requires changes. The required effort varies largely based on where the test is being run.

In case the tester has a local environment at their disposal, the change might be very quick, as the GUI application is already installed and the files exist in the system. If the tester needs to reserve a UTE environment, the files need to be imported to the Flame application, after which the changes can implemented. This can be a time consuming process.

Another option is to run the test normally from the UTE, however, this requires editing the test files on the tester's personal computer, after which they need to upload the file into the file repository in order for the UTE to be able to fetch the correct file for the test run. The time used for this often depends on the merging speed of the Git repository, which may vary within the time-of-day.

Test creation with Flame (UTE)

When creating a new test case for Flame to be run in UTE, the tester needs to first decide the general configuration the test should run in. The wanted configuration is given to the Flame environment settings and to the UTE reservation for the automation to work correctly, as shown in Figure 8. The environment settings are saved into a separate file and can be downloaded to any other test that is run in a similar test environment. The environment setting -file is saved to Git along with the test settings -file.


Environment selection (FRGU_ASIL_3A1G_BTS_Settings.ecs)

Feedback: bts_ietv_ta_support.nsn@nokia.com

RF/Opt switchs settings
Configuration settings

LBTS-1 (CLOUD_FR4L_L_A_[11]_[FF
Cleware (10.35.233.35)

Site name in settings: CLOUD_FR4L_L_A_[11]_[FF

Settings from tag
UTE Cloud tag: CLOUD_FR4L_L_A_[11
Cells: [3A1G]

Site information
```
Site name:
CLOUD_FR4L_L_A_[11]_
[FRGU_FRGU]_[ALD]_
[3A1G]
Cabinet type: Flexi
BTS
Cell count:   4
Unit count:   10
```

☑ Use local connection
☑ Single BTS OAM
☑ Use BTS connection for IM

Port used with IM conn: 3600
Reference testline:

Cells

Local ID	Global ID
1	1
2	2
3	3
4	4

Figure 8. Screenshot of Flame environment settings. When preparing UTE tests, the environment settings need to include the used topology.

After preparing the environment settings or downloading them from an existing file, the tester has to install or update the test scripts. After this, the test steps are defined by using the Flame drag&drop feature to select the correct steps from a list. The test steps are very high level, and the minimum requirement is to choose the test itself, but often the tester wants to use additional steps to perform looping, configuration changes, etc.

The test is finished by choosing the correct settings for the test, after which it can be saved to Git and the test run can be started in UTE.

3.4.2. Robot Framework

Robot Framework (RF) was originally developed at Nokia but was later released as an open-source automation testing framework [12]. Since it is an open source -project, it can be modified to many different purposes [13][10]. Robot Framework is widely used for web testing purposes, but in this project it is used for testing BTS SW.

RF has a special feature, where it saves class instances and other variables to a cache, so they can be used in all files without making separate imports. This feature is usable also inside Python files by using RF's BuiltIn -library, making it very simple to fetch the once found information inside all used files.

Robot Framework is a keyword-driven test framework, which makes the test structured and thus it produces an orderly HyperText Markup Language (HTML) -report, which is easy to read and provides a general overview of the events during the test.

However, the orderly manner for executing the test makes it impossible to have multiple tasks happening at the same time, so it was necessary to use Python threading for creating listeners to observe the BTS behaviour throughout the test instead[14].

While listener threads are making the test faster as there are no "dummy" waits, it also has a downside to it. Unlike in Flame, the Robot listener is constantly polling the BTS, which sometimes results in significant delays in BTS activities, for example, when downloading new software to the BTS. In SW download tests, the times were observed to be even 6x longer than they should have been according to specification. This lead to the decision to stopping the listeners/polling for the duration of some test steps in order to prevent the automation from affecting the test result.

```
*** Settings ***
Documentation
...   Author:   Programmer X

Resource    testsuite/../OMKPI/Robot/Resources/OMKPI_common.robot

*** Variables ***
# Build that is active in the beginning of test
${PATH_TO_BUILD}
${OLDER_SW_BUILD}

*** Keywords ***
Upload Swconfig And Reset site
    Run Keyword And Continue On Failure    Upload File To Bts
    Run Keyword And Continue On Failure    Reset Site
   detailed_output=True
    Run Keyword And Continue On Failure    Wait For All Cells Up
   time_limit=360
    [Teardown]    Run Keyword If  '${KEYWORD STATUS}' == 'FAIL'  Set
    Suite Variable  ${KW_SUCCESS}    ${False}
```

Listing 3.1. An example of a keyword file for a Robot test

```
*** Settings ***
Documentation
...  Author:   Tester Y

// Library and Resource imports here

Suite Setup       Common Suite Setup
Suite Teardown    Common Suite Teardown

Test Setup        Run Keywords  Common Test Setup
...               AND           Start Fault Notification Listener  $
   {ALLOWED_FAULT_FILE}
Test Teardown     Common Test Teardown

*** Variables ***
${wait}                  3 min
${path_to_sw_build}=    testsuite/.../sw_build.zip
&{SMODS_AND_PORTS}       SMOD1=9
&{EXPECTED_FAULT}        fault_id=${4082}
${ALLOWED_FAULT_FILE}   testsuite/.../allowed_faults.txt

*** Test Cases ***
Short Press And New SW
    [Documentation]   Make short service button press, collect logs
   from Rescue Console, upload and activate new SW and
    ...    perform commissioning with scf file.
    Run Keyword In Loop And Return Pass Rate
     ...    name=Short Press And New SW
     ...    test_case=3
     ...    loops=1
     ...    keywords=Short Press And New SW
     ...    expected_pass_rate=1
```

Listing 3.2. An example of the test file for Robot - edited by the tester.

During the design process for the Robot testing, the main focus was making the usage of the scripts as easily understandable as possible for people with little to no experience with programming. As shown in Listing 3.1, the steps defined in Keywords should make sense to a non-programmer, as it is mainly plain English. To familiarize the testers to all available keywords (KW), in addition to the example test cases, a documentation is generated from the code.

The keyword file is mainly written by the automation group, while the test file is done and edited by a tester. An example of a test file is shown in Listing 3.2. The structure of a file is simple. The settings and Variables are given first, then the Keywords and/or Test Cases. By separating the Keyword and Test Case files, an additional benefit to the ones mentioned before is the length of each file, which is kept shorter when they are separate from each other. Although the KWs are in different files, they are easy to track with the help of an Integrated Development Environment (IDE), such as PyCharm. In the Keyword -file, the test keywords are given under Keywords. The test file then calls the keyword in line 29. Once a variable is initialized in a Robot -file, it can be called at any level of the file hierarchy.

Robot Framework itself is not very flexible in terms of taking into account lots of scenarios for the different BTS configurations with different behaviours, unless they are done within separate Robot -files. However, this can be overcome by handling the parts requiring a more dynamic approach within the Python code underneath RF.

Therefore, the needed dynamicity for analyzing the BTS behaviour is done with Python, along with plenty of more complex parts that require the function or method to be dynamic, while RF acts more as a facade on top of the Python code. This way of using Robot Framework also provides the ease of use the testers need to create their own tests without the help of test automation developers, so they can use their resources to more complex tasks.

As the automation becomes more dynamic in regards to different BTS configurations, it allows the usage of one test in any test environment, with only a few limitations. This allows minimizing the amount of test files in the file repository, and decreases the time used in editing files as there are less files to begin with. The more dynamic approach also removes the need to make separate files for different SW branches that are under testing, which can reduce the needed files by even 3-4 times.

The keywords used in the robot test are defined in libraries made by our team or Test Automation Framework -team. TAF -library mainly provides keywords that are usable within the entire testing organization. The keywords are designed mainly for performing simple tasks with the BTS API, for example, sending a site reset request.

Our libraries are more focused on providing support for the more specific activities. As an example, the TAF -library may provide the KW for sending a power reset to a radio unit, however, collecting and parsing the required information of the specific radio unit is done using our automation library, as well as the checks that come later on.

Special testing requirements

While using RF, it is easy to edit the variables from the command line, overriding the variables in the robot file. This is a useful feature when there is a need to make temporary changes to the variables both Robot Framework and Python code are using.

The command line changes are also supported on the UTE side, wherein it is possible to give a specific addition to the command to an optional text field while starting the test run. It is also simple to perform changes to the test when reserving the UTE environment, as the files do not need to be imported to an application once the tester receives the environment for themselves.

This is an especially useful feature when it comes to special testing, as there is the possibility for the tester to only give the command on the command line without making changes to the files. These variable change possibilities cover, for example, if there is a need to change the log collection or waiting times, or informing the TA system about special logging HW attached to the BTS.

Albeit the variable changes are easy to perform, it requires an additional skill from the tester, as they need to be able to understand how to use the command line efficiently. To overcome any lack of competence within this area, the automation team is organizing trainings for the rest of the team whenever it seems necessary, as well as collecting a guide with information from regular testing and environment setup to special testing.

Test creation with Robot (UTE)

Creating a new test for Robot requires a text editor, but it is easier after setting up an IDE. The tester's responsibility is to create the high level test file, which is shown in Listing 3.2.

The tester can duplicate the file from the existing ones, and change the Documentation (line 3), variable values and possibly add or remove some variables, depending on the test case (lines 15-19). The test KW can be changed on line 22 and it is the one UTE finds when setting up the test reservation. The tester is recommended to make a test description to test case documentation on lines 23-24. The test name is given on line 26. The test case has a number on line 27, and it can be greater than 1 in case the test file consists of multiple test cases. The number of loops the test is run is given on line 28. The test KW is given on line 29, and it can be found from the KW file. If the test has multiple loops and it is enough for a certain percentage of them to pass, the tester can give the success rate in line 30, for example, if it is enough that 80% of tests pass for the test set to pass, expected pass rate has a value of 0.8. The file is then saved to Git, and can be run in UTE.

4. IMPLEMENTATION

The work consisted of implementing the keyword libraries to support the team needs and creating example test cases. In the beginning, we were studying the syntax of Robot Framework, which can be simple at its best, but there were also many things that we learned during the process of building the new framework.

Some later learnings forced us to change the KW methods on Python side, for example, learning how RF saves the variables to a cache, removing the need to import the variables from/within RF -files to Python -files/other robot -files. This, in the end, makes the files simpler as there is no need to import all variables from upper level files to lower levels. As shown in section 3.4.2, the test file includes all variables. Any variable introduced in the test file can be used in the keyword file without importing the variables separately, or in Python -files using get_variable_value(<variable>) from RF's BuiltIn -library.

4.1. Planning

As the first phase of planning, I made the design for our git repository, to make sure everything finds a logical place instead of the files ending up in random folders or all contributors using their own logic. To begin with, the folders were divided between all the testing areas and in addition, one Common -folder was prepared for all the code that is used in all test areas.

Each folder then has a sub-folder for Robot and Python files. This separation was done mostly to separate the files made by scripting team (Python) and testers (Robot). Furthermore, the Python folders include Interfaces- and Resources -folders, to separate the files by usage.

Interfaces -folder includes files that can be used in Robot -files, while Resources includes files that are used only from other Python -files. This helps the testers separate the purpose of each file, whenever they are looking for KWs they can use in their tests. To further support the testers, a documentation is created and kept up-to-date using a Python library pydoc.

The Robot folders have more variety in structure, as they were mostly planned within each test area, allowing the testers the freedom to design the folders based on their individual needs. The Robot -folder under Common includes Keyword -files, which are used within all test areas. One example of such a file is for keywords to perform a configuration change.

- **Common**
  - Python
    - Interfaces
    - Resources
  - Robot
    - Resources
    - TestKeywords
- **Test area folder**
  - Python
    - Interfaces
    - Resources
  - Robot
    - ExampleTestCases
    - TestKeywords
    - OtherFolders

In conclusion, the file system is designed in a way that the work done by testers and TA developers are separated, in order to make a clear distinction of who can edit the files and the responsibilities are clear for all.

4.2. Implementation

The implementation was started by forming a connection to the BTS and setting up log collectors. In addition to the HTML -log that is automatically collected by RF, we also prepared an event log, a simple text log that shows the events in a chronological order.

The most significant difference to the RF's HTML -log is seen when the fault listeners are making prints to the event log, as they are seen together with other prints. In the HTML -log, the prints are in a different level, which makes the analysis more difficult. RF's HTML -log is, however, more useful in error analysis and checking the overall success of a test, which makes both logs useful in the analysis process, as they complement each other.

The implementation was started by preparing files that are required in the pre-requirements for a test, collecting the configuration information from the BTS. The connection itself was easily done using the TAF -libraries. The algorithm for iterating the Configuration Identifiers was brought from Flame, which required translating a few C# -files to Python.

The configuration collection is a part of every test, which led us to use a Common Resources -Robot file that does the preparations and post-activities automatically when the given KWs are called. Instead of the tester having to start all the listeners and checks in the beginning of the test, it is enough to call one KW instead. This makes the test files much cleaner, easy to read and less prone for typos or other mistakes.

After the pre- and post-condition KWs were prepared and tested, the first simple tests were made. The TAF -libraries support most of the activities we needed to do to the BTS. The tests are often more complicated than checking the pre-conditions, performing an activity and then checking the post-conditions, which gives us the need to build our own checks and often also additional test KWs.

Majority of the work has consisted of parsing the correct information from the json's we get from the BTS and sending messages via SSH connections.

4.2.1. Timeline

The project was started in the fall of 2021, with the help of our colleagues. They were offering us training and support wherever it was required, and we began to assign resources to the project development. In the beginning, I did most of the preparations and planning, as the rest of the automation team was busy with Flame.

The first thing was to learn to use the Robot Framework, along with the existing TAF libraries that were built by our colleagues. The first test environments were prepared for testing, and the first simple test was done in the beginning of 2022. Throughout the year, the rest of the base work was done for the pre- and post-condition checks, as well as log collection. During the year, the automation team was able to assign more resources into the project and the project was then progressing faster.

By the end of the year, fault notification listeners and the more common testing scripts were done, to provide the base to creating official tests, providing us the chance to start involving testers in the process to give feedback on the logging and the system.

From the beginning of 2023, the plan was to start creating tests at an increasing rate to replace the Flame tests in the official testing. For the first quarter, the aim was to have 5 tests in the testing with the test result being reported officially. The first tests were the simplest ones, that had only one action between the pre- and post-condition checks. The purpose was to mainly test the performance of the steps that would be present in all of the test cases.

In the end of Q1, the first tests were done, tested and reported. At this point, the work was done by 2/3 programmers in the team, to allow enough resources to supporting Flame as the main automation tool. I was the only one working on the project 100% from the beginning of 2023, but throughout the development of the Robot project, more resources were transferred to the Robot development from Flame support.

Between Q2-Q4, a little over 50% of the tests were moved from Flame to Robot. During this time, more testers were involved in the project and by the end of the year, all testers had at least some experience with the Robot automation system, some with all their tests already running solely with Robot.

The project is scheduled to be finished by the end of May 2024 due to Flame tool support ending then, which puts pressure on the initial schedule where the project was scheduled to be done by the end of year 2024. This timeline challenge is partly being tackled by the testers, who are reviewing their tests and reducing the overall tests wherever it is reasonable.

4.2.2. Challenges

Creating an easy-to-use system

Over the project, we faced several challenges, the first one being, how to plan the system in a way that it is easy for the tester to use and do as much work on their own as possible. This required a lot of trial and error, ultimately resulting in a few layers of test files. The first test file is filled with different tests, often being similar in nature. It can be written by the tester, in case they are more confident in using Robot Framework, or by the automation team.

The second, or top, layer of the test file includes a simple structure that is repeated in every test file. It includes only the test KW and the variables, making it easy for the tester to make changes to the test variables without seeing the more detailed test structure underneath. Although it is not a GUI, this top layer test file looks very simple and is meant to be easy to learn to use.

The top level file should only be edited by the tester, and during the project the structure has experienced many versions and alterations before it reached its current form. At this point, the same structure is usable in all testing areas, to keep it simple and make it easier for testers to offer support to each other also between different testing areas.

Learning to use the environment files in UTE

Over time, it also became clear that there will be too many test files in case the test environment variables need to be given in the test file. During the second half of 2022, we discovered the use of test environment files in UTE, which allowed us to make the Robot system more dynamic. There was no longer a need for the tester to provide the environment information to the test, even with special equipment.

Since the environment file is very handy in testing, a similar approach was applied to the local environments. The responsible tester needs to fill the environment information, such as power breaker IP addresses, to the file in a specific format and save the file to a predefined location. This reduced the amount of code that is needed to search for the required information, as the format of the information remains very similar in both UTE and local test environments.

Working with two operating systems

In the end of 2022 and during 2023, the operating system of the test computer was also a challenge. TAF does not support Windows, so initially the idea was to create a VM image in Linux, that could be used on top of Windows. However, over time, there were too many problems with this, starting from slowness, to memory problems, to the occasional crashing of the VM.

All UTE computers or VM's use Linux as their operating system, which meant the problem was only with our local test environments. Due to this, it was decided that our system would not support Windows either, and the OS of each computer needs to be changed to Linux when the tester wants to start using Robot. Since Flame supports both, Linux and Windows, the change does not affect testing with Flame, in case the tester wants to keep it as a backup during the transformation.

Resource division in the end of the project

As the project is now at a stage where a little over 70% of the tests are finished, and the timeline has been tightened, we are facing the challenge of resource sharing. The tests that are left to be done, include difficult challenges that have not been solved yet, and creating working solutions might be time consuming.

Supporting two separate automation systems at the same time requires a lot of resources.. A limited time is left for the development of new tests, as the existing tests are also requiring their own time whenever there is a need of fixing bugs, assisting the testers in various situations and improving the general system, whilst maintaining support in Flame for the leftover tests.

Due to these challenges, we are facing slower development time, with 40% of the time in our hands compared to the year 2023 as the schedule was tightened. In an attempt to tackle the resourcing challenge, we started putting more focus on prioritizing our tasks, doing only the necessary ones for us to reach our goals and putting the ones on hold that will improve the usability, but do not affect the transformation process.

4.3. Repositories

With the wider transition from Flame to Robot, the automation files were decided to be saved into a different repository with stricter rules in what should be put there, with more overlook on the files. Flame automation files are saved to a test repository called RobotLTE, and the new repository is called RobotWS. Their contents vary a lot, as well as the structure within the /testsuite because of differences in automation system requirements.

Due to Flame's own limitations, each software branch needs their own file, as the reporting identifier is given to the Flame test case settings, hence the amount of files is larger compared to those required for Robot automation system. RobotLTE is a much older repository, originally created in around the year 2013 for UTE purposes, whilst RobotWS is a much newer repository, originally meant for 5G testing and later harnessed for Robot automation purposes, supporting testing from 2G to 5G tests.

RobotLTE has increased in size massively due to very little upkeep, which has resulted in many obsolete files and folders in the repository. RobotWS is overlooked more carefully, as it is constantly supervised by a group of people referred to as Code Guards, who are responsible for keeping an eye on the quality and necessity of the files and folders in the repository. Both repositories also have automatic pipelines, but RobotLTE pipeline does not do many checks, whilst RobotWS pipeline checks for code quality using PEP-8 standard and confirming the Python syntax is correct, as well as checking that all imported files exist. Over time, this way of working will hopefully result in a smaller repository size.

TAF keeps their own Robot libraries on their own repository, to which it is also possible for users to create user libraries. User libraries are not supported by TAF, but the one who created the code. The purpose of creating user libraries is to share the code with other teams, if there exists such a need. However, adding ones library to the user library -collection might mean more work in supporting other teams in addition to ensuring their own team's needs are met, which is not always possible or reasonable.

RobotLTE is bigger in size as it has a larger variety of contents, which can be seen in 9. The size has increased, for example, because the user libraries are located in the same repository as other automation files, instead of, for example, the TAF library repository.

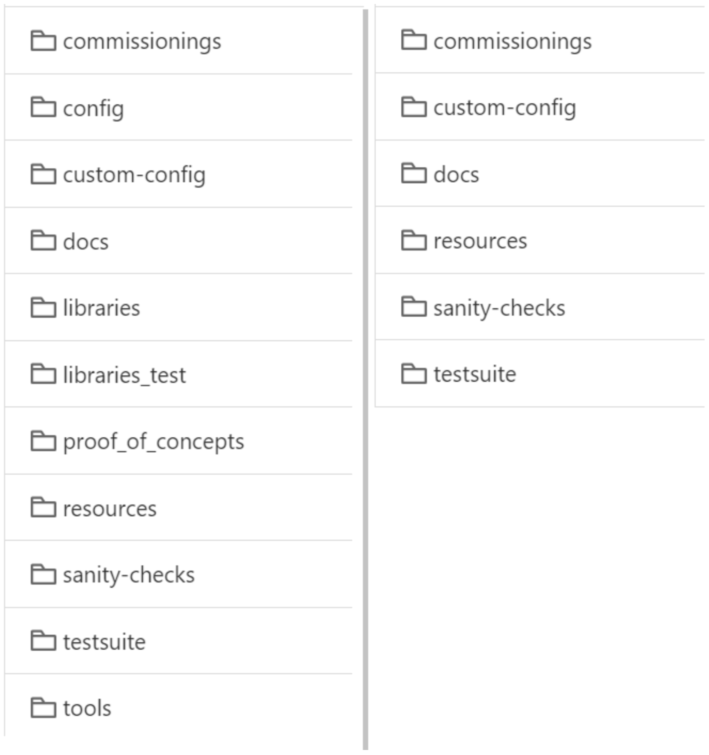


Figure 9. Main folder level of each repository. RobotLTE is on the left, and RobotWS on the right.

RobotWS is used in a different manner compared to RobotLTE. There are no user libraries as in RobotLTE, and each team's library is its own repository and they are packaged separately. Therefore, it is not possible and is forbidden to refer to files outside of each team's library, which has its ups and downs. In RobotWS the option is to create a separate user library when another teams wishes to use the code of others or create copies of the needed code into their own library.

Whilst there might be more work needed to provide a user library or duplicate code between different teams' libraries, it is also easier to separate the responsible teams of each piece of code and they are better kept up-to-date due to the Code Guards - approach.

Keeping each teams code separate from each other, the merging pipelines are faster, as it is not necessary to check all code from the entire repository, but to check only the edited code and its effects within that team's repository.

To further improve the repository user experience, a Gears project is aiming to create entire separate repositories within the repository. Each team's repository could have their own dependencies to the TAF - or user libraries, which could make the pipeline times during merging even faster, as there is no need to check all possible library compatibilities, only the ones that are really used within each team's code.

5. USABILITY EVALUATION

The purpose of the evaluation is to assess the usability of each framework. The evaluation is focused on the efficiency of creating and editing tests, assessing the log quality by considering the ease of creating fault reports, as it is done based on logs, and by the ease of understanding the issue based on logs.

The results will most likely experience some bias, as most testers have over 5 years of experience with Flame, whilst only the duration of this project with Robot Framework, at most. The Robot -project is still in progress and not nearly as mature a tool as Flame, which will also affect the results. However, the evaluation will give insights to the potential that lies within the Robot -TA system, once it matures over time.

5.1. Usability Questionnaire

The usability of each framework is measured using the responses to a questionnaire, which is discussed in more detail in Chapter 5.2. Usability is considered from two different perspectives: ease of use and efficiency. The tools are compared based on the time used for analyzing and reporting issues, as well as how independently the tester can perform a problem analysis without the help of others, most often the scripting team.

Along with the ease-of-use, we are also looking into the previous experience in programming languages and how that affects the tester's ability to adapt to the Robot system, as it often requires working with code to a certain extent, as well as sending commands from the command line. Since Robot Framework is a keyword-driven framework, it is expected that it can be easily understood even without prior programming experience. Additionally, the programming efforts from the tester side are minimized, as described in Chapter 3.4.2.

As discussed in Chapter 4.3, the Git repositories are different for each tool, and their structures are also evaluated in the user experience questionnaire to compare how easy it is to find the correct files, as well as how fast the changes are merged. Both affect the user experience, as the changes can be implemented faster and testing can start sooner.

5.1.1. Questionnaire Design

The questionnaire is done using Microsoft Forms, where the questions are divided into five (5) sections. Since all participants are native Finnish people, the questionnaire is conducted in Finnish to prevent as many misunderstandings as possible due to the language used.

All sections are highlighting that the tester should only consider UTE testing in their responses, as it is the primary test environment in use. Sections 2 and 3 are copies of each other to provide comparable results for the two systems.

Section 1 - Programming skills

1. How familiar were you with programming before the start of the project? *(F: Kuinka tuttua ohjelmointi oli sinulle ennen projektin aloitusta?)*

2. If you responded something other than "Not at all" to 1, what programming languages did you understand/knew? *(F: Jos vastasit ylempään jotain muuta kuin "Ei lainkaan", mitä kieliä ymmärsit/osasit kirjoittaa?)*

3. Have you learned Robot Framework (and Python) during the project? *(F: Oletko oppinut Robottia (ja mahdollisesti Pythonia) projektin aikana?)*

Section 2 - Flame

4. How easy to use do you generally experience Flame? *(F: Kuinka helpoksi koet Flamen käytön yleisesti?)*

5. How much time do you estimate to use when creating a new test? (incl. Settings preparation, script package updates, building the test, changing the variables and updating to Git) *(F: Kuinka paljon käytät arviolta aikaa uuden testin luomiseen? (sis. Asetusten valmistelu, skriptipakettien päivitys, testin rakennus, muuttujien säätö ja päivitys gittiin))*

6. How much time do you estimate to spend doing temporary (special testing) changes? For example logging changes, wait time changes or loop number changes. This does not include the time that is used for setting up special logging equipment in a laboratory. *(F: Kuinka paljon käytät arviolta aikaa väliaikaisten (erikoistestauksen) muutosten tekoon? Esimerkiksi logituksen muutos, odotusaikojen muutos tai loop muutos. Tähän ei lasketa aikaa joka kuluu erikoislogien asennukseen labrassa)*

7. How easy it is for you to understand the logs without consulting your colleagues? *(F: Kuinka helppo sinun on ymmärtää logeja ilman että kysyt neuvoa tulkitsemiseen muilta?)*

8. Free comment section for complementing the responses to above questions or sharing user experience.

Section 3 - Robot

9. How easy to use do you generally experience Robot? *(F: Kuinka helpoksi koet Robotin käytön yleisesti?)*

10. How much time do you estimate to use when creating a new test? (incl. Writing the test or copying an old one or an example test, changing variables and updating to Git) *(F: Kuinka paljon käytät arviolta aikaa uuden testin luomiseen? (sis. Testin kirjoitus/kopio vanhasta tai esimerkkitestistä, muuttujien säätö ja päivitys Gittiin))*

11. How much time do you estimate to spend doing temporary (special testing) changes? For example logging changes, wait time changes or loop number changes. This does not include the time that is used for setting up special logging equipment in a laboratory. *(F: Kuinka paljon käytät arviolta aikaa väliaikaisten (erikoistestauksen) muutosten tekoon? Esimerkiksi logituksen muutos, odotusaikojen muutos tai loop muutos. Tähän ei lasketa aikaa joka kuluu erikoislogien asennukseen labrassa)*

12. How easy it is for you to understand the logs without consulting your colleagues? *(F: Kuinka helppo sinun on ymmärtää logeja ilman että kysyt neuvoa tulkitsemiseen muilta?)*

13. Free comment section for complementing the responses to above questions or sharing user experience.

Section 5 - Creating a fault report

14. Is either tool easier to write a fault report with? If yes, why? For example, ease/difficulty of finding the required logs, ease/difficulty of interpreting the logs, writing the fault report description/analysis based on the logs,.. *(F: Onko jommastakummasta työkalusta helpompi tai vaikeampi kirjoittaa pronto? Jos kyllä, miksi? Esimerkiksi oikeiden logien löytämisen helppous/vaikeus, logien tulkinnan helppous/vaikeus, prontokuvauksen kirjoitus logituksen pohjalta,..)*

Section 6 - Git repositories

15. Which repository is clearer when considering the structure? In other words, from which one it is easier to find the test/files you need. *(F: Kumpi repository on selkeämpi käyttää rakenteeltaan? Toisin sanoen, helpompi löytää hakemasi testi.)*

16. Is either one faster to make changes? *(F: Onko jompaankumpaan nopeampi tehdä muutoksia?)*

17. Additional comment (voluntary).

5.2. Results

The team has 13 testers, all responded to the questionnaire. 10/13 testers have been testing with Flame for over 5 years.

The results are presented as Likert scales wherever applicable, with average and Student's t-test values, all of which were gotten using Microsoft Excel. The data samples are paired, and the null hypotheses for the t-test is there is no difference between the systems, and the alternative hypothesis suggests there is.

The prior experience with programming is represented in Figure 10. More than 50% (7 out of 13) of the testers are able to read and write code to a certain extent, but the experience is with other programming languages. As seen in Figure 11, the experience is focused on other programming languages (8 out of 13) and Python (6 out of 13), while RF was known by only 2 out of 13 testers before the project started. However, the learning process is easier when there is any background in programming, as the programming logic remains the same.

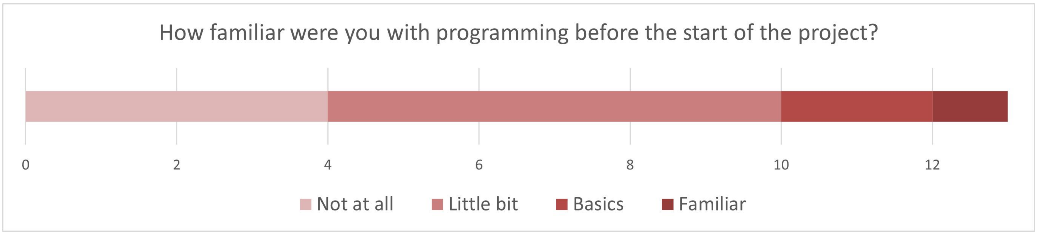


Figure 10. Question 1: Programming experience before the project started: 4/13 had no prior experience, 1/13 had full proficiency. 6/13 were able to make minor changes and 2/13 could read code.

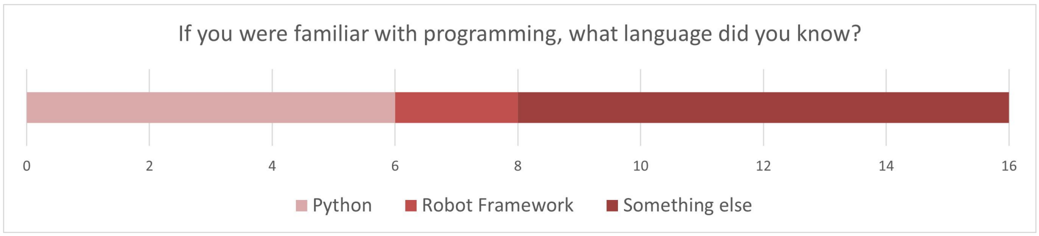


Figure 11. Question 2: Those with experience in programming, the experience is focused on Python (6/13) and other programming languages (8/13).

As seen in Figure 12, only one tester reports they did not learn any programming during the two years of the project, which indicates that there has been at least some competence improvements as a side benefit to the project.

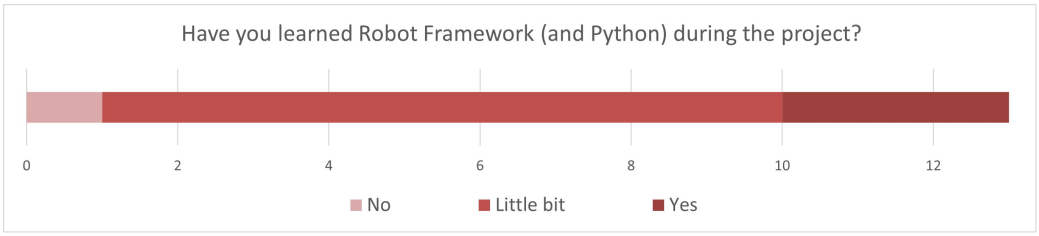


Figure 12. Question 3: Programming competence development over the project duration. 9/13 responded they have learned at least a little bit, 3/13 experienced they have learned new programming skills during the project.

The free comments provided further insight into the users' experience with each aspect of testing. Flame requires more efforts due to the fixed configurations in each test, creating the need to make multiple files for the same test ("A Flame test is tied to one configuration after it is done. If you want to test with different cell or radio amount, you need to make a new test.", P1). It was also experienced slow to prepare ("The environment preparation takes a lot of time; prepare Flame, download script packages, import test file from your own PC and before that get changes from Git. Flame itself is very slow and crashing often, especially when I'm handling bigger tests.", P2). However, it was considered very easy to use due to the GUI ("Flame is very simple to use", P4).

In comparison, Robot is benefiting from its ability to adjust to all configurations, removing the need for multiple test files per test. ("Robot test scripts are easy to run in different configurations without making any changes. Variables can be changed from the command line, it is extremely convenient and fast.", P1; "I need to do lots of testing for fault reports (incl. special testing) and my days are mostly filled with them. It takes only a few minutes to make the needed changes and run the test in the terminal. ... There is no need to transfer files between computers, like in Flame. ...", P2). Even though the overall test results show that making temporary changes is slower compared to Flame, some say it is very fast to change variables from command line.

Creating a fault report was a similar experience for 5 out of 13 respondents, while Flame was considered easier by 5 out of 13 respondents due to it being more familiar and more advanced due to the logging being further developed in comparison to Robot.

However, Robot was considered easier by 3 out of 13 respondents due to clear logs and smaller log sizes when considering BTS logs, allowing them to use the fault reporting tool in a more simplified manner as it has a size limitation for attachments ("...Robot logs are simpler and shorter, only Event log is currently too large...", P2; "Robot usage is easy and it is good that it is being developed.", P6).

Robot system also got compliments by the newer members of the team as being easier to learn when they had no prior experience with Flame ("As a person who was learning Robot and Flame simultaneously, I like Robot better. Fast and simple and less dragging things from one place to another.", P2).

5.2.1. Ease-Of-Use

Ease-of-use is measured subjectively by collecting data on how the testers experience the usage of each test automation system. One part of the ease-of-use is how the testing is experienced generally, and another point of view is the easiness of reading and understanding the logs that differ based on the chosen automation system.

The general user experience in each TA system is mostly affected by the existence of a GUI. Because of Flame's GUI, it does not have the same requirement of understanding programming as the Robot TA system.

Ease-of-use is considered in questions 4&7 from Flame PoV, 9&12 from Robot PoV and in question 15, which is focused on the repository structures. The testers are asked to estimate on a numerical scale from 1-7 how easy it is to use the system generally.

Secondly, the testers are asked to assess how easy it is to read the logs in order to create an analysis of the test run. This has a major effect on the system user experience, as good logs make a big difference in the work of a tester. As stated in Chapter 3.4.2, the only difference to each automation system is the event log that is written by the TA and Robot Framework is additionally providing an HTML log.

Thirdly, a factor in ease-of-use is how the Git repositories are designed and how easily the user can find the files they are looking for. The data to review this is collected in question 15.

Figures 13, 14 and 15 are representing the results for ease-of-use questions.

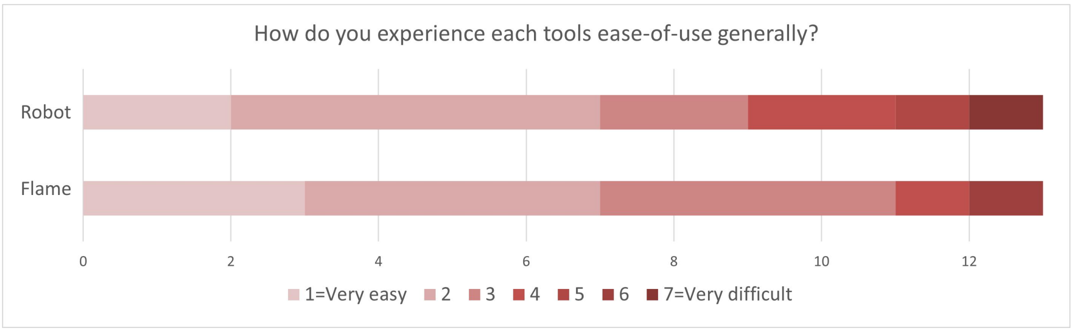


Figure 13. Questions 4&9: Comparison of how easy the testers experience the tools to be used.

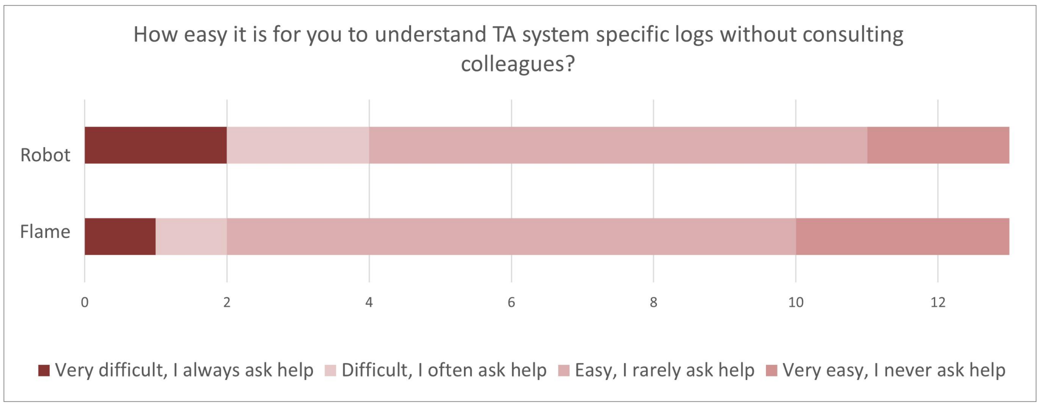


Figure 14. Questions 7&12: Comparison for log readability in terms of understanding the BTS and automation behaviour without consulting colleagues.

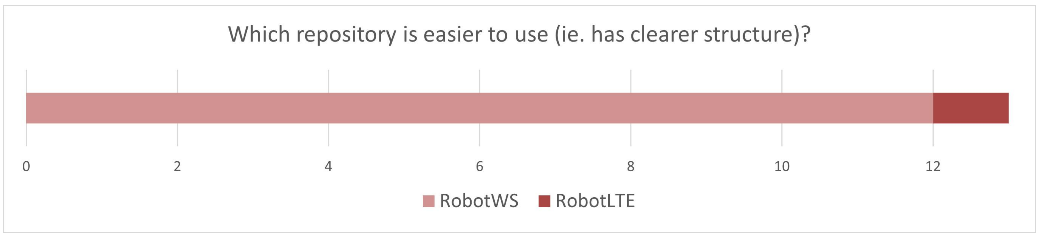


Figure 15. Question 15: Comparison between the two Git repositories and their structures, regarding how easy they are to navigate and find the correct files. RobotLTE is used for Flame and RobotWS for the Robot project.

Flame is experienced as easier to use, with Flame at an average of 2.54 (1=very easy, 7=very difficult) and Robot at 2.92 (p=0.299 using Student's t-test). In Figure 13 it can be seen that while most users are comfortable with both systems, Robot is experienced by some users users as difficult or very difficult. This could be explained with some users being involved in the project for a longer period of time, whilst some are still preparing their first tests for Robot, thus lacking the experience that would improve the ease-of-use experience, as they get more familiar with a different way of testing.

Numerically, the difference is not big and the gap will likely close over time, as the Robot TA system is finessed and the testers gain more experience, especially those who have not been testing much with Robot system by the time the questionnaire was performed.

When comparing the log readability, the Flame specific logs are considered to be slightly easier to read, at an average of 3.00, while Robot specific logs were given a grade of 2.69 (p=0.109 using Student's t-test). This implies there are still improvement needs for Robot logging, but the difference at this point is quite small, considering the state of the project.

Looking at Figure 14, Robot system logs are experienced as very difficult to read by some testers, but the experience is mostly very similar to the one users have with Flame. This experience will improve over time while the logging is improved, as well as when the users gain more experience and understanding of the system.

In the text answers, the main problem with Robot logs is that the HTML log consists of much more information compared to that of Flames. It is not necessarily a bad thing, as it also provides more insight to the system and what results the BTS is sending to the automation system. This will possibly become a more positive feature also for the tester, once it becomes more familiar to them and they learn what to look for. From a scripting point of view, the information is very useful and often requires less debugging in problem situations, making the system development and bug fixing process faster.

The repository usage was considered in Question 15, that is represented in Figure 15. As discussed in Chapter 4.3, the RobotWS is better up-kept and the structure is very different, with less folders for each testing area, mainly due to the automation system requirements. This is very clearly visible in the question results, where RobotWS is experienced as easier to use by 12/13 users, compared to RobotLTE. However, in the free comments, it was also mentioned that the time it takes with merging and pipelines varies a lot in both repositories, which makes the usage unpredictable ("The times vary a lot. There is no way of knowing if the merge will be fast or slow.", P5).

5.2.2. Efficiency

Efficiency is also measured subjectively by combining the numerical values for the estimations given by testers in the questionnaire regarding the time it takes to prepare new tests or editing old ones for special testing purposes. The purpose is to see if there are any experienced time savings in the new system, however, as stated before, this will also likely face some bias as it takes time to learn the new system and use it as fast as possible.

Especially special testing is expected to be faster in Robot system, as it is often not required to make any file changes when the settings can be changed using variables in the robot -file. This does not cover all cases of special log testing, but it should create a noticeable difference in the time used.

The Flame efficiency is evaluated in questions 5&6, Robot efficiency in 10&11 and the repository efficiency in question 16. Each automation system's efficiency is considered from two perspectives - how fast it is to create a new test (Q5&10) and how fast it is to make minor changes to the test for special testing (Q6&11).

It is important to notice, that the questionnaire does not evaluate the time used to making changes to the test files when a new software branch is taken into testing or when a new configuration is introduced, as these are only related to Flame testing. However, it is still an important factor that should be considered when comparing the two frameworks in order to evaluate their efficiencies.

Figures 16, 17 and 18 show the results for questions related to evaluating the efficiency of each automation system and their repositories.

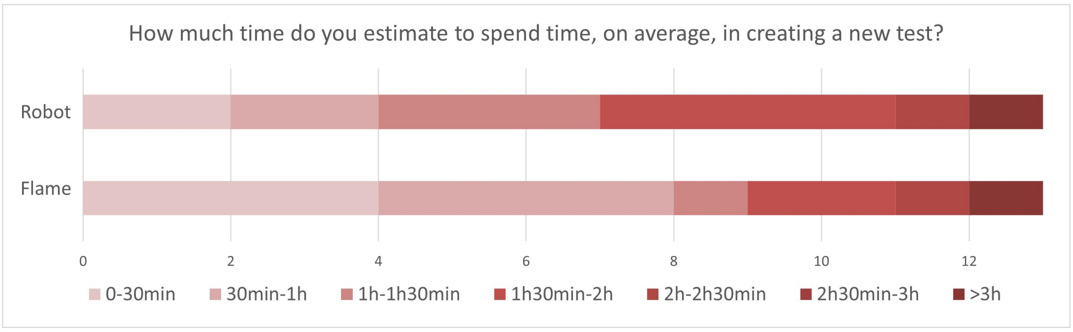


Figure 16. Questions 5&10: Estimation of how much time the tester uses in new test creation on average.

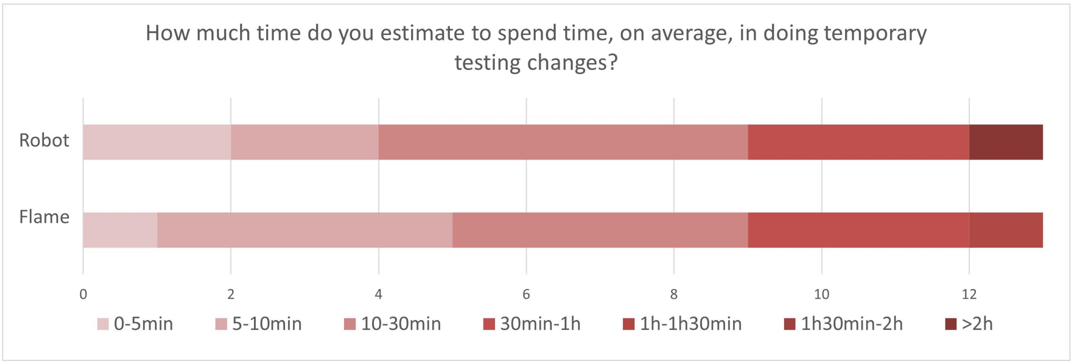


Figure 17. Questions 6&11: Estimation of how much time the tester uses to make the relevant changes for special testing on average.

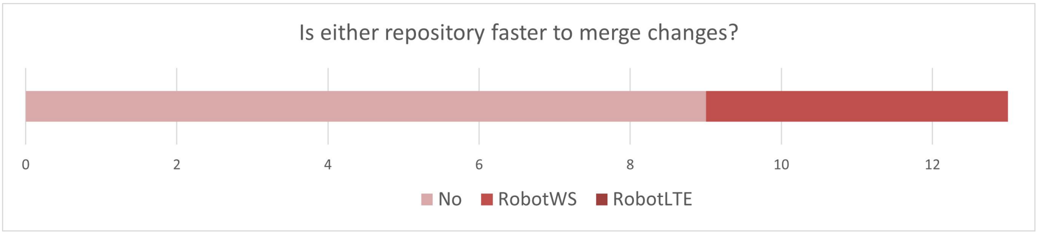


Figure 18. Question 16: Comparison between the two Git repositories and effect they have on the time that is spent to merge the changes. 4/13 claim RobotWS is faster, while 9/13 claim they are equally fast.

The time used in creating a new test and editing the test also gives an indication of how easy the system is to use, thus making efficiency and ease-of-use closely related to each other. The average time to create a new test in Flame was evaluated at 2.69 (2=30min-1h,3=1h-1h30min), whilst for Robot the time was evaluated at 3.31 (3=1h-1h30min, 4=1h30min-2h) (p=0.199 using Student's t-test).

From Figure 16 we can see, that the new test creation time is less than 1 hour for 8/13 testers, while for Robot the time used is between 1 hour to 2 hours for most testers. From the text based results, the Flame benefits mostly from the GUI, which makes the system very easy to use, and thus also faster. However, it is occasionally experiencing slowness and due to each test being tied to a specific configuration, Flame requires more tests to be done overall.

When it comes to making temporary changes for special testing, Flame had again shorter time estimation at an average result of 2.92, while Robot is estimated at 3.08 (2=5-10min, 3=10-30min) (p=0.388 using Student's t-test). The difference is small, but statistically relevant according to the t-test, and possibly indicates further training needs for Robot usage. As discussed earlier in Chapter 3.4.2, the temporary changes can be made from command line or giving the variables in the UTE reservation, indicating that most changes can be doable within minutes. The need for lengthier changes is rare, and often done by scripting team instead of the tester.

Figure 18 shows that the merging times are experienced to be the same, or RobotWS being faster. The merging pipelines will hopefully become faster along the Gears -project, which should limit the amount of scripts that the pipelines check during the request, thus improving the user experience even further and making the work more efficient.

6. ANALYSIS

The answers to each tool's questions from the questionnaire are compared based on the multiple choice answers, while taking into consideration also the optional open text -answers. The collected data is represented as Likert scales, comparing Robot and Flame results wherever applicable. It is expected that the user experience will still change over time as the testers get more familiar with the new system in comparison to Flame, from which most testers have over 5 years of experience.

During the 2,5 years I have worked with this project and since the beginning of year 2023, I have held a meeting bi-weekly to introduce the new framework to the testers and hear their thoughts, problems and ideas. During that time period, most testers started practicing to use the new automation system and have been providing valuable feedback on how to develop the automation system.

After using Robot for more than a month, the feedback received in the meetings became more positive compared to the beginning of the project. The positive feedback is mostly focusing on the ease of adding more configurations into the testing scope, which was also seen as a major benefit in the questionnaire results.

Other positives that have come up in the meetings have been the ease of editing a test case and the testing for fault analysis being simple with the variable changes on command line, as described in Chapter 3.4.2. However, the questionnaire showed more variance in how these were perceived. Editing a test case is experienced as a time consuming process, however, the answers are very similar. Only one response indicated the test edit was to take over 2 hours in Robot, which may also be an indication of lack of experience and knowledge in the system. On the other hand, it also shows that the system is not as intuitive for all users, as some are taking longer to familiarize themselves with it and to becoming more effective and more comfortable in using the automation system.

The Robot automation system has increased the amount of different test environment configurations per test case as the system is very dynamic and can manage any test environment without making changes to the test files. This results to lower maintenance resource needs, allowing more time to other tasks. For the developers, the workload does not appear to be much higher, as the various configurations have been taken account previously as well. The difference to the old system is from where the configuration details are fetched.

Overall, Flame is still more liked and more efficient in the aspects that were observed in the questionnaire. However, even with longer test creation times, the Robot system has reached a more dynamic state regarding the different configurations and SW versions, reducing the amount of situations where a new test needs to be done. This will quickly even out the time used with the tests itself, offering the testers more time in designing their tests and problem investigation.

The questionnaire only scratches the surface of how the two systems' usability differ, and how the used repositories affect the user experience. The usability is only considered regarding the efficiency and general ease-of-use based on the testers . The efficiency is considered in new test creation and for temporary changes, however, the results cannot be the only thing that should be looked at. In the open comments, it was noted that Flame is more tied to the configuration, requiring more new test files in comparison to Robot. This can result in more time spent with the test case files overall, as the changes or new tests are needed in larger quantities.

Robot is still a new system for a lot of its users. The newer testers with less experience on Flame are reporting that the change has been a very easy process and the Robot system has been perceived as easier to learn in comparison to Flame. For this reason, it is also possible that the users with more experience in Flame usage, may also experience some bias when considering any new system, thus making the questionnaire results biased as well. The lack of GUI may not be the only reason to resisting the automation system change, one factor may simply be the difficulty of adapting to change itself.

Overall, the questionnaire gives a good indication of how the system is working and what needs to be improved. Especially the logging improvements would drastically improve the system usability, because the testers' work largely evolves around the logs and understanding the BTS behaviour based on them. When the automation provides a good analysis in its logs, that are easy to read and understand, it is also easier to locate the issue within the BTS logs. Good automation logs also save a lot of time, if they are able to correctly point the tester to the right direction when investigating the root cause of a problem.

7. DISCUSSION & CONCLUSION

This thesis describes the process of changing the automation system from the old system Flame to a new Robot system using Robot Framework. The discussion begins from testing in general, different methods used in testing and how the automation can make the processes more efficient, offering a wider test scope with less resources.

For the test automation system to work as expected, it is important to understand the system under test (SUT), and the relevant tools that are used with it. In this project, the BTS is the SUT, and the important tools such as UTE and special testing equipment, such as special logging collectors and unit power breakers are described. When designing the automation system, it is also important to know how the SUT will be tested and what are the expectations to it by different stakeholders, especially the testers as they will be the primary user.

The major improvement from the old system was the dynamic nature of the Robot system regarding different test environment configurations, leading to less manual work required by tester when creating or changing test files.

The change from a new automation system to another is a time consuming project, and the time required to meet the user expectations can be high. To measure the success of the new system from user perspective, a questionnaire was performed after the testers had experience with the new system, spanning from a few months to over a year.

The questionnaire results were showing Flame outperforming Robot in both aspects, efficiency and ease-of-use. However, the results may be showing some bias due to the vast experience the testers have in Flame testing, whilst Robot being a new system for all. Both of these factors have an effect on the results of the questionnaire, but it is difficult to estimate the extent.

RQ1: How does the ease-of-use differ between each automation system?

The usage of Flame and Robot TA systems are very different from each other due to Flame having its own GUI, whilst Robot is run from the command line. This difference is, however, not having much effect in the actual testing, as the UTE scheduler does the work in regression testing, which covers most of the testing done. New test creation and test edits are more affected, as the tester needs to read and edit text files when using Robot, instead of using the drag-and-drop functionality of Flame's GUI. Hence the questionnaire also focused on these two aspects to evaluate the user experience the testers have with the automation systems.

At the time the questionnaire was performed, the testers' answers indicate that the overall usage and ease-of-use of Flame is easier, due to the GUI, but it also was experienced to be slower due to the actual application crashing often and being generally slow, in addition to the need of updating the scripts and having to manually enter test environment information to the system. The log readability was experienced as quite similar, Robot having slightly worse result.

RQ2: How efficient it is to use each system when creating new tests or making changes to existing ones?

Overall, Robot is not far behind Flame's usability and it will improve over time, as the system will be further developed and the users gain more experience with it. Changing an automation system is not an easy nor a fast process. Making the transition smooth, a lot of preparative work and training is required to make the change as painless as possible for all stakeholders.

The new test creation was experienced only a little bit more efficient in Flame, while temporary changes had a worse rating. However, this is contradictory when considering that most temporary changes are very easy to do from the command line when running the test, not needing any changes to the files. This led to keeping a training session to ensure all users are familiar with this possibility. After the training, the result might have been very different to the one that was received in the performed questionnaire.

Considering the users' experience and the maturity difference in each system, the Robot shows potential in becoming a more versatile tool for the automation, offering multiple improvements to the testing, such as easy and fast special/temporary testing and its dynamic nature when considering different configurations, which has resulted in more found faults as more configurations have been in use.

8. REFERENCES

[1] IBM What is Software Testing. URL: https://www.ibm.com/topics/software-testing.

[2] Hamilton T. (2023) Manual Testing Tutorial: What is, Types, Concepts. URL: https://www.guru99.com/manual-testing.html.

[3] European Union (2022) Infographic - Energy crisis: Three EU-coordinated measures to cut down bills. URL: https://www.consilium.europa.eu/en/infographics/eu-measures-to-cut-down-energy-bills/.

[4] Nokia (2020) Nokia dynamic spectrum sharing for rapid 5G coverage rollout. URL: https://onestore.nokia.com/asset/207265.

[5] Nokia (2020) Nokia announces first phase of its new strategy, changes to operating model and Group Leadership Team. URL: https://www.nokia.com/about-us/news/releases/2020/10/29/nokia-announces-first-phase-of-its-new-strategy-changes-to-operating-model-and-group-leadership-team/.

[6] Nokia (2023) 6G explained. URL: https://www.nokia.com/about-us/newsroom/articles/6g-explained/.

[7] Pip - Python package installer. URL: https://pypi.org/project/pip/.

[8] Jenkins. URL: https://www.jenkins.io/.

[9] Stresnjak S. & Hocenski Z. (2011) Usage of robot framework in automation of functional test regression .

[10] Bisht S. (2013) Robot Framework Test Automation. Community Experience Distilled, Packt Publishing.

[11] Khan E. (2011) Different approaches to black box testing technique for finding errors .

[12] Robot Framework ry Robot Framework - Introduction. URL: https://robotframework.org/.

[13] Open Source Initiative. URL: https://opensource.org/.

[14] Python Software Foundation threading - Thread-based parallelism. URL: https://docs.python.org/3/library/threading.html.