FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Ville Kivikko

TOOLS FOR DETECTION AND ANALYSIS OF FLAKY SOFTWARE TESTS

Master's Thesis
Degree Programme in Computer Science and Engineering
December 2022

Kivikko V. (2022) Tools for Detection and Analysis of Flaky Software Tests. University of Oulu, Degree Programme in Computer Science and Engineering, 60 p.

ABSTRACT

Software testing is an essential part of developing a high-quality product. Test cases that pass or fail in a nondeterministic manner can cause severe problems and be difficult to fix. Unstable test cases cause increased resource usage on many different levels, as well as delays for the project they appear on. Increased resource usage might also cause delays over the project boundaries. There are multiple different reasons why tests might become unstable. A common one is the order dependency between test cases, which can also make the addition of new test cases more difficult.

At the start of this work, the test cases of an under development version of eCPRI module were rather unstable. To improve the stability level, this work introduces three tools designed for test cases based on CppUTest framework with multiple use cases to help the developers to tackle the issue of unstable test cases occurring due to order dependencies. The tools are aimed to investigate different types of order dependencies that could occur between test cases. The main methods the tools are utilizing are test repeating and shuffling the testing order.

In some cases, the tools were able to increase the reproducibility of the occurring failures, which is very important when solving the root cause for the failures. The tools also provide an easy way to perform an extensive, automated test running which offers a chance for the user to be absent while the logs are gathered from multiple test runs, which can be helpful when investigating random failures. Combined with the principle of binary search on reducing the executed test cases, the usage of these tools was useful when pinpointing the tests and reasons for failures. The stability level of the eCPRI module's test cases was improved with the methods and tools presented in this thesis. The stability level of shuffled test runs was brought up from 0 % to 70 % and the stability level of normal test order was brought from 82.5 % to 100 % for one of the compilers used.

Keywords: CppUTest, order dependency, repeating, resource leak, shuffling, test stability

Kivikko V. (2022) Työkaluja epästabiilien ohjelmistotestien tunnistamiseen ja analysointiin. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 60 s.

TIIVISTELMÄ

Ohjelmistotestaus on välttämätön osa korkealaatuisen tuotteen kehitystä. Testit, joiden lopputulos vaihtelee epämääräisesti, voivat aiheuttaa vaikeita ongelmia ja ne voivat olla hankalia korjattavia. Epästabiilit testit kasvattavat resurssien käyttöä useilla eri tavoilla, minkä lisäksi ne aiheuttavat viivästyksiä projektin etenemiseen. Kasvanut resurssien käyttöaste voi myös aiheuttaa viivästyksiä toisiin meneillä oleviin projekteihin. On useita syitä miksi testistä voi tulla epästabiili. Yksi yleisimmistä syistä on testien riippuvuus niiden suorittamisjärjestyksestä, joka voi myös hankaloittaa uusien testien lisäämistä.

Työn alkaessa kehitteillä olevan eCPRI moduulin testit olivat suhteellisen epästabiileja. Osittaisena ratkaisuna tähän ongelmaan tässä työssä esitellään kolme työkalua, jotka on suunniteltu helpottamaan eCPRI moduulin CppUTest sovelluskehystä hyödyntävien testien stabiiliuden parantamista. Työkalut keskittyvät erityyppisten testien välillä olevien järjestysriippuvuuksien tunnistamiseen ja paikantamiseen. Tärkeimmät metodit, joita työkalut käyttävät, ovat testien toistaminen ja testijärjestyksen sekoittaminen.

Työkaluilla onnistuttiin parantamaan joidenkin ilmenneiden suoritusvirheiden toistettavuutta, joka on äärimmäisen tärkeää paikannettaessa virheen aiheuttajaa. Työkalut myös tarjoavat helpon tavan automatisoidun laajamittaisen testauksen suorittamiseen, mikä mahdollistaa useiden testiajojen logien keräämisen ilman tarvetta jatkuvalle käyttäjän läsnäololle. Tämä helpottaa tärkeiden tietojen keräämistä ja kokoamista harvoin tapahtuvista testi- tai suoritusvirheistä. Näiden työkalujen käyttö, yhdistettynä puolitushaun periaatteeseen ennen virhettä suoritettujen testien määrän vähentämisessä, oli hyödyllistä virheiden aiheuttavien testien ja syiden paikantamisessa. Tässä työssä esitettyjen työkalujen ja metodien käyttämisellä onnistuttiin parantamaan eCPRI moduulin testien stabiilisuutta. Sekoitettujen testiajojen stabiilisuusaste oli työn alkaessa 0 % ja se onnistuttiin nostamaan 70 %:iin. Tavallisen testijärjestyksen stabiilisuusaste oli työn alussa 82.5 % ja se onnistuttiin nostamaan 100 %:iin.

Avainsanat: CppUTest, järjestysriippuvuus, resurssivuoto, sekoittaminen, testien stabiilisuus, toistaminen

TABLE OF CONTENTS

FOREWORD

I am grateful for the opportunity to complete this thesis for Nokia and would like to thank everyone in the team for supporting me in this journey, especially Juho Stenudd and Virpi Hanni, for following up the process actively and providing good insights.

I would also like to express my gratitude to my supervisors Pekka Sangi and Olli Silvén for providing me valuable guidance throughout the process of completing this thesis.

The support I have received from the people who are close to me has also been really valuable, and I would like to thank them all for that.

Oulu, December 6th, 2022

Ville Kivikko

LIST OF ABBREVIATIONS

3GPP	The 3rd Generation Partnership Project
4G	Fourth Generation
5G	Fifth Generation
BBU	Baseband Unit
CI	Continuous Integration
CPRI	Common Public Radio Interface
CU	Central Unit
DL	Downlink
DU	Distributed Unit
eCPRI	enhanced Common Public Radio Interface
FH	Fronthaul
gNB	gNodeB
L1	Layer 1
L2	Layer 2
LLS	Low Layer Split
LTE	Long Term Evolution
MAC	Medium Access Control
MT	Module Test
NR	New Radio
OBSAI	Open Base Station Architecture Initiative
ORI	Open Radio Interface
O-RAN	Open Radio Access Network
PBCH	Physical Broadcast Channel
PDCCH	Physical Downlink Control Channel
PDSCH	Physical Downlink Shared Channel
PRACH	Physical Random Access Channel
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
RAN	Radio Access Network
RF	Radio Frequency
RRU	Remote Radio Unit
RU	Radio Unit
SUT	System Under Testing
SoC	System on Chip
TDD	Test Driven Development
UE	User Equipment
UL	Uplink
UT	Unit Test

1. INTRODUCTION

Extensive software testing is an essential part of successful software development. Unit tests are meant to verify the software one small piece at a time to ensure that the software functions correctly [1]. It also prevents developing code that is buggy and has low quality, as the units of code that provide incorrect output are noticed easily through test failures. It is shown that having high code coverage leads to fewer failures after release [2], as more of the program gets tested in the testing phase. However, high coverage percent is not a direct indicator of the quality of the software, as the quality of the tests is an important factor as well [1].

High code coverage can be achieved for example by using test-driven development (TDD), which is a widely used software development practice where unit tests are developed before the product code the unit test actually tests [3]. Using TDD often leads to a commonly known Red-Green-Refactor cycle, where Red refers to developing automated tests that are expected to fail, because the product code does not exist yet. Green refers to making those tests pass by developing the code forward and Refactor refers to cleaning the code after the Green has been reached [4].

This cycle often happens also when some already released product, or a product which is still under development but is already developed further, is taken in use as a basis for a newer product. This is because a lot of the already existing tests can be used for that product as well. Even if this is the case, the developers must be certain that everything important is tested thoroughly, so that no bugs would be left till the end product.

Tests can be considered as polluting if they do not clean things they changed or if they do not release the resources they used while testing [5]. If some test leaves some flag changed for example, the next test might be written the way that it presumes that the flag is in the state the previous test left it. State-polluting tests are the main reason for unwanted dependencies between tests [5]. If the previous test gets disabled or changed at some point, or the test execution order gets changed, the latter test starts to fail, because the state is not what it assumes [6]. Memory leaks can be hard to detect if the usage of the memory is not monitored [7], and they can cause crashes at later point of the software testing.

In an ideal situation, the tests would pass even if they are run in any order or even as many times as wanted with a single start up, if there are no limitations set for that. Overlooking the cleanliness of unit tests can lead to flaky test cases, crashes, bugs in the tests or production code, and will most likely cause delays and extra work. Flaky test cases are tests that pass or fail in a nondeterministic manner [6], and they can cause problems of a different kind.

In Layer 1 Fronthaul Software (L1 FH SW) development in Nokia, many different products that are currently under development use the same code repository. The products which have similarities in some modules can use parts of the existing code and test cases if they are suitable for both. Using previous accomplishments again is called software reuse, which usually leads to improvements in SW quality and productivity [8].

The same code repository leads to using the same Continuous Integration (CI) pipeline. CI is a practice for software development, where the developers who use the same repository integrate their updates into the code frequently. The integration

is verified by multiple automated builds by the CI pipeline to detect possible errors quickly, so that they are not merged to the master repository [9 chapter 1].

CI can be configured with different ways, depending on the project. For example, broader regression testing can be done after the merge to speed up the process, but also to ensure that the changes do not accidentally break other projects. If some tests are unstable, and they are marked as blocking ones in the CI pipeline, work gets delayed because the pipeline reports a failure and either stops the run or starts it again. Rerunning failed jobs is in use for this project, which is a way to handle unstable test cases, but it can also hide some instabilities as well as actual bugs. This leads to queues in CI pipelines since the tests are executed again, which leads to larger resource usage. This can lead to big delays especially if the pipeline has many users, because the users have to wait for a long time before their changes can get tested properly to be merged to the master branch.

In the situation before this thesis, there were a lot of unstable tests in this completely new version of Enhanced Common Public Radio Interface (eCPRI) [10] module developed for a new product. These instabilities had required a lot of time from the team responsible for the module. It had also slowed down the development for other teams through the errors occurring in the common CI pipeline. The goal of this thesis was to introduce tools that help to locate the unstable module tests, figure out the reasons why the tests fail and thereby improve the stability of the eCPRI module.

This thesis is structured in the following way. Chapter 2 presents the working area, Fifth Generation (5G) L1 FH SW, and the eCPRI module in it. Chapter 3 presents some theory about common testing practises concerning this thesis and Chapter 4 goes through some theory of flaky tests. The situation and background to this thesis are represented in Chapter 5, and the tools created to improve the stability of this module are presented in Chapter 6. The usage of those tools is presented in Chapter 7. Discussion and conclusions of this thesis takes places in Chapters 8 and 9.

2. WORKING AREA

This chapter provides a short overview of the area this work focuses on. The main areas are introduced with a reasonable level. Due to the main idea of the thesis is not tied into the working area, the technologies of the working area are not introduced with very technical and detailed level.

2.1. 5G New Radio

As the world is constantly becoming more and more connected and devices and services are constantly developing and offering more to the customers, the requirements for the network connection get more demanding as well. Fourth Generation Long Term Evolution (4G LTE) is not capable of meeting these requirements much longer, or it is already too slow, depending on the requirements set for the application. Due to this, it is obvious that network connectivity needs to develop as well. With 5G New Radio (NR), first specification approved in 2017 in Release 15 by The 3rd Generation Partnership Project (3GPP) [11], these requirements can be tackled, and mobile network connectivity can achieve outstanding performances which cannot be compared with any earlier solution. Three main cornerstones to tackle by 5G are extreme mobile broadband to acquire faster data rates everywhere, massive Internet of Things communication and ultra-reliable low latency communications for instant action [12 p. 3].

Based on the 3GPP 5G NR Radio Access Network (RAN) architecture model, the functionalities of the gNodeB (gNB) base station, which is a successor to eNodeB used with 4G LTE, is divided into two parts: Distributed Unit (DU) and Central Unit (CU). However, depending on the chosen functional split that were introduced by Open Radio Access Network (O-RAN) Alliance, Radio Unit (RU) can be separated from DU, and it can be used to perform some functions for gNB, making the functionalities divided into three parts. The gNB architecture is illustrated in Fig. 1

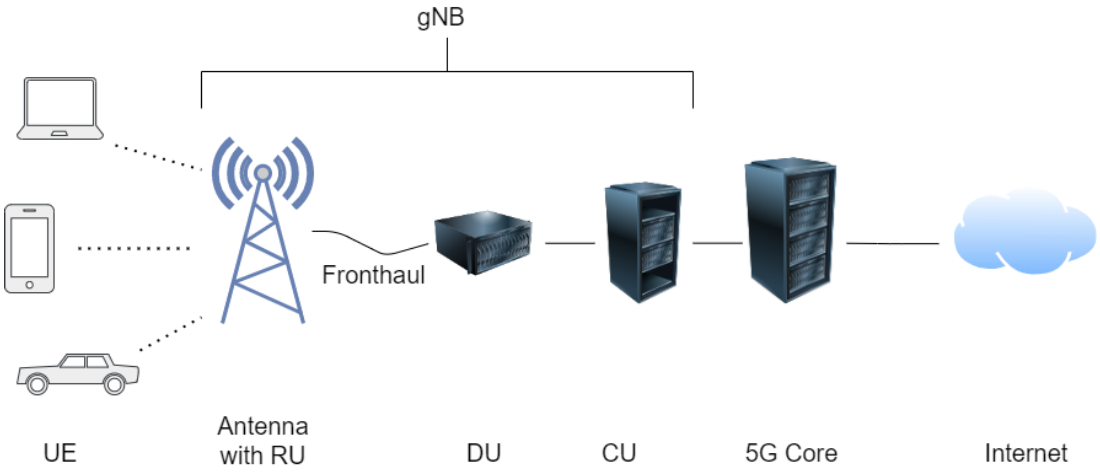

Figure 1. Simplified architecture of 5G with gNB.

Currently, performance records for 5G NR are 8.08 Gbps for downlink (DL), acquired by Samsung [13] and 2.1 Gbps for uplink (UL), acquired by Nokia [14]. When developing a system that is desired to break these high performance records, every module in it needs to be constantly performance tested to ensure that the changes made to it do not decrease the performance levels unnecessarily. This emphasizes the importance of software testing starting early in the developing process.

2.2. Physical Layer

Layer 1 (L1), the physical layer, traditionally locates in the DU. However, depending on the functional splits some L1 functions can be done on the RU side as well. Traditional layer split is illustrated in Fig. 2 below. The physical layer has many responsibilities in 5G NR, such as timing control and synchronization for UL, random access procedures, channel coding and multiplexing, and multi-antenna mapping for example [15]. In 5G NR, L1 has six different physical channels, which have different responsibilities. These channels and their main responsibilities are presented below [16].

- PBCH (Physical Broadcast Channel) – transmits device access related system information
- PDCCH (Physical Downlink Control Channel) – carries control information for DL
- PUCCH (Physical Uplink Control Channel) – carries control information for UL
- PDSCH (Physical Downlink Shared Channel) – carries user data for DL
- PUSCH (Physical Uplink Shared Channel) – carries user data for UL
- PRACH (Physical Random Access Channel) – handles the random access

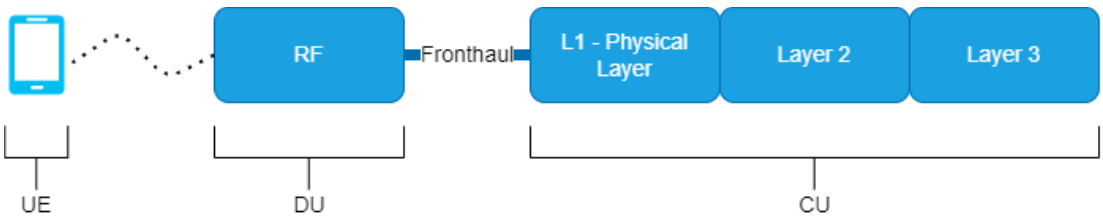

Figure 2. High level illustration of traditional functional splits with layers.

As L1 locates between Radio Frequency (RF) block and Layer 2 (L2) in the 5G NR RAN architecture, its job for DL is to receive transport blocks coming from L2 – or more precisely from L2's Medium Access Control (MAC) sublayer locating at the bottom of the L2 – and doing the correct procedures for the transport blocks in question and then to pass them on towards the RF block [17 p. 197-199]. For UL, the job is to do the conversion the opposite way: take in data from the RF block, do correct

procedures depending on what type of data the input contains, and create a transport block to be passed on to L2's MAC sublayer. Therefore, transport blocks function as input to DL and output for UL for the physical layer [17 p. 218-220].

2.3. Fronthaul Interface

FH is the interface between DU and CU with the high layer split defined by 3GPP, and if O-RAN Alliances Low Layer Split (LLS) is taken in use, it is between RU and DU. When regarding a 4G system, FH is between Baseband Unit (BBU) and Remote Radio Unit (RRU). FH can be implemented with four different interface options: Common Public Radio Interface (CPRI), eCPRI, Open Base Station Architecture Initiative (OBSAI) or Open Radio Interface (ORI) [18]. As eCPRI is founded by the O-RAN Alliance which consists of the biggest vendors in the field – and it has different advantages such as being suitable with different Options and it can co-exist with CPRI [19] – it is likely that eCPRI will be taken in use as FH protocol among different 5G devices from different vendors.

The layer related location of FH depends on the chosen functional split. 3GPP Option 8 is the traditional split which was used with 4G LTE where FH was between RRU and BBU. With gNB, in Option 8 the FH is located precisely between L1 and RF block, only leaving RF functionalities on the DU side. In Option 7, the FH locates inside the physical layer L1, which is split into two different parts by the FH: Low-PHY and High-PHY. Option 6 places the FH between L1 and L2, which means that all the L1 functions are done in the DU. Lower Option number means more functions performed in the DU, and less in CU [20].

O-RAN Alliance takes the splitting further, by introducing the LLS, which splits the DU into two parts: RU and DU. This enables even more different options for vendors to decide from. The LLS offers a chance to divide the gNB into three physical parts, where the connection between RU and DU is done with FH, and DU is connected to CU with Midhaul. The chosen option has huge effects on the outcome of the product, as it defines what is implemented in which part of the gNB. This has effects on the requirements for FH's performance and many different things on the whole gNB level, such as the performance, energy consumption and the costs to name a few [20].

Taking the 3GPP Options 2 and 8 as examples, the Option 8 would provide a small DU with fewer costs, as most of the functions are performed by CU, as illustrated in Fig. 3. That would then demand very high performance from FH and would cause some disadvantages, such as high requirements on FH latency [20]. Option 2 would mean large and demanding DU with high energy consumption as most of the things are done in the DU, but that would require less from FH. If O-RAN Alliance suggested Options are taken in use, the three-way split offers a chance of locating those physical devices into different locations, which opens different options on how to handle the maintenance, updates and the power consumption of those devices for example.

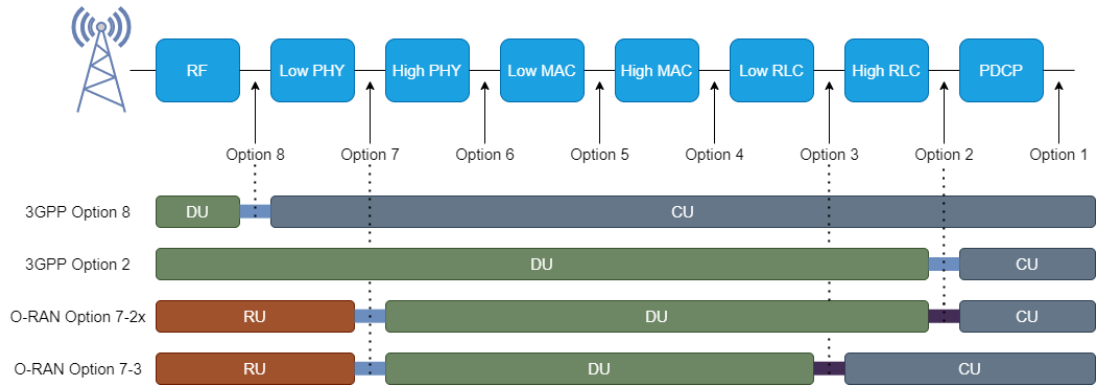

Figure 3. Options for functional splits proposed by 3GPP and O-RAN with couple of examples.

2.4. Enhanced Common Public Radio Interface

The eCPRI interface is an enhanced version of CPRI and can be used as FH. It was introduced to tackle some constraints existing with CPRI for 5G products. While 4G LTE has a fixed slot length of 1 ms, in 5G NR the slot length varies from 0.0625 ms to 1 ms [21]. CPRI is more limited, as it can be used with Option 8 only, while eCPRI has many different Options available, what affects the requirements and performance hugely. eCPRI is also more energy sufficient, as CPRI's bandwidth utilization is not dictated by the mobile payload, which makes it waste resources when the load is not high. eCPRI offers packet-based FH [12 p.76-78] what makes data rates be dependent on the utilization of the resource blocks at all times. Packet based FH also enables the usage of Ethernet as connection, instead of dark fiber which is used with CPRI [12 p.76-78]. Due to the fact that eCPRI and CPRI follows the same standards, they can be present in the same system together [22].

When compared with the earlier alternatives, eCPRI has several advantages to the base station design. It can reduce the required bandwidth about 10 times and it is flexibly scaled according to the user plane traffic. The chance of using Ethernet as the connection offers the chance to carry eCPRI traffic as well as other traffic in the same switched network simultaneously. eCPRI also enables the use of sophisticated coordination algorithms to ensure the best possible performance, due to it being a real-time traffic interface. The eCPRI interface is also future proof since it allows the introduction of new features through software updates with a radio equipment controller which is located in DU/CU when low layer split is in use, without the need for updating the radio equipment itself, which is at the antenna [23].

When comparing the estimates for data throughput requirements for the FH interface between different physical layer splits, the Option 8, using CPRI, requires 236 Gbps throughput rate, while Option 7, using eCPRI, reduces the throughput requirements massively, only requiring 14-30 Gbps. With Option 6 even more procedures are done in the DU, so the throughput requirement becomes even lower, reducing it to as low as below 4 Gbps [10].

The eCPRI interface is responsible for creating and handling Control and Management plane messages for UL and User plane messages for UL and DL. Using

some O-RAN split option, the messages are sent from DU to RU or from RU to DU depending on whether the transmission direction is DL or UL. Control and Management plane messages are used to pass information needed to control the system properly. User plane messages carry the user information from the base station to the user equipment when the direction is DL, and vice versa with UL [10]. While User plane messages are considered being time critical, Control and Management plane messages are not [10]. In order to meet the accuracy and timing requirements, some handlers and timers must exist inside the eCPRI module.

3. TESTING PRACTICES

If test-driven development is taken in use in the project, the majority of the tests are written before the code. As unit tests are written earlier than the actual code, designing the tests makes developers to think the software in smaller parts. This can help the developers to understand the software and its purposes more clearly [3]. However, in big software projects – and in projects that are reusing some existing software – there might be different people designing the unit tests than developing the software itself. The possibly gathered deeper understanding from developing the tests might not be on the current developers. However, if the tests are existing and correct, the developers can gain knowledge from the tests by examining what the functions under development need to do and what aspects need to be taken into account to produce correct output.

Developing embedded software and tests for it, the actual product hardware might not be available in the beginning of the SW development. This often leads to a situation that the tests are developed for the host, simulated System on Chip (SoC), or for some previous, already existing product HW that can be used to test some parts of the software. It might also be beneficial if the existing HW can be used to simulate the upcoming product HW as good as possible.

The testing process for an embedded system proceeds beforehand and together with the actual development process. Testing starts from unit testing (UT) and then proceeds to module testing (MT) if modules and units are separated by design, which can depend on the complexity of the product and/or practices used by the company developing the system. After that, integration testing takes place, where multiple modules or units that are connected to each other are tested together to verify whether they function correctly together. Next steps are validation and system testing. In validation testing, testing is designed to cover all modules and functionalities, verify their correct functionality together with their performance, and to validate that against the requirements set for them [7]. In system testing, the system is tested thoroughly as it would be running in the real situation it will be used. The last step is acceptance testing, where the system is tested against the requirements set for it, to see if the full realization of the system meets the initial purpose of it and to see if its performance is on required level. These testing phases are illustrated in Fig. 4.

When developing a module with very strict requirements, regression testing becomes even more important than usual. Regression testing is a testing practice, where the program gets tested after every modification, to ensure that it continues to perform correctly according to specifications set for it [24].

3.1. Continuous Integration

Continuous Integration (CI) is a practice in software development, where the developers integrate the changes they are proposing frequently to a shared repository through a common pipeline [9]. Without CI practice, the integration of developers' work in big software projects would most likely require much more manual work and adjustments from the developers.

The test procedures that are run by CI is called the CI pipeline. It is an enabler for the possibility of fast integration to the master branch for developers' work. The CI

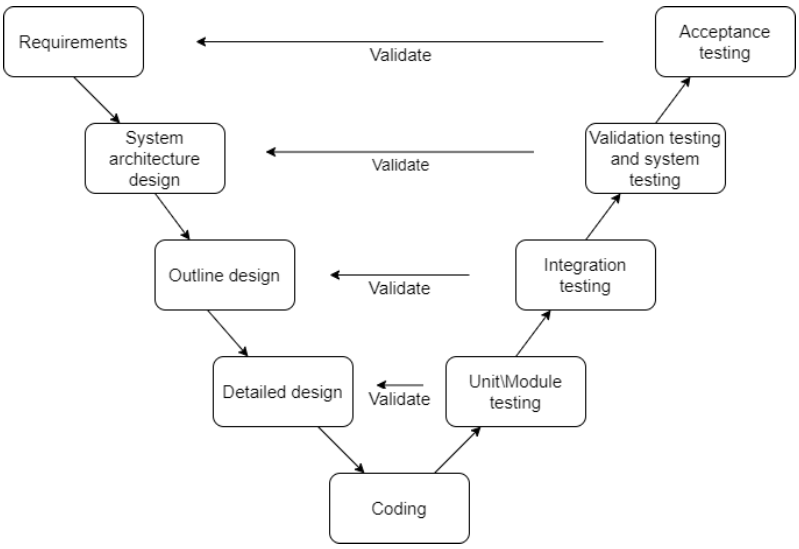

Figure 4. V-model for developing embedded systems presenting the scope of different test phases [7].

pipeline can be configured to automatically do checks of a different kind and builds for the software project is used on. Automated test executing done by the CI pipeline helps enormously with regression testing. Traditional functions and procedures for a CI pipeline are [9]:

1. Triggering a build every time when commit appears
2. Executing existing unit and module tests automatically
3. Building possible artifacts, for example docker or reference values
4. Executing existing acceptance tests
5. If everything OK, merging the changes to the master repository

If some procedures configured to the pipeline fail, the changes will not get merged, the user is notified from the existing error and if logs are available, they are derived for the developers by the pipeline. The CI pipeline can also be configured to run some tests after a successful result from the pipeline. If there are existing tests that have an unacceptable level of stability for example, they can be marked as non-blocking tests in the pipeline, but still be executed to gather more information about them and to see if some change has effect on them.

3.2. Unit Testing

Unit testing (UT) is, as the name suggests, testing small units from the software. The unit to be tested can be some function for example. Testing that unit can be done at the simplest by comparing the outcome of the function with a known result for the given parameters. The purpose of unit tests is to bring up any misbehaviour of the software,

and to act as a key enabler for sustainable progress of the software project in question. A software project with no unit tests can get a flying start for the project, but when the project grows and gets more complex, the progress often slows down when bugs start to appear, which is illustrated in Fig. 5. It is shown that unit tests improve the quality of the code. When developing unit tests from the start of the project or even beforehand as TDD suggests, the progress for the project starts later, but is not as likely to end up getting blocked by bugs as it would be without unit testing [1].

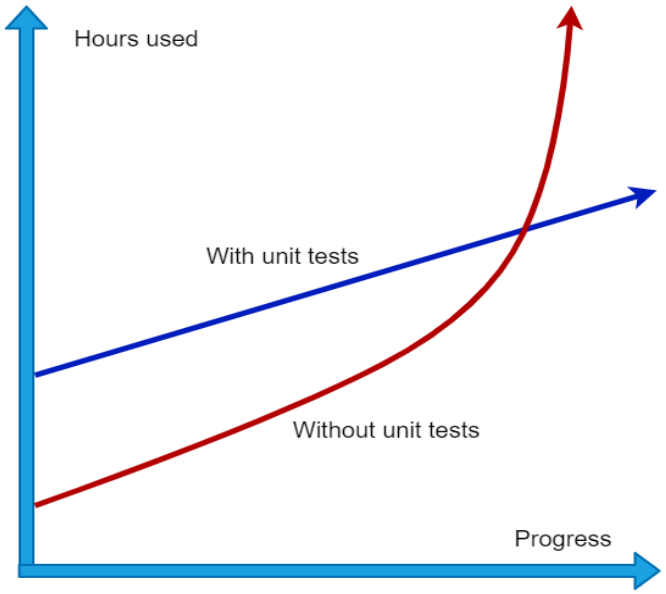

Figure 5. Spent hours in relation to progress in a typical software development process, with and without unit tests [1].

3.3. Module Testing

Modules, also referred as components, are parts of a bigger system that are independent, but act together to form this bigger system. Dividing the bigger system under development to smaller logical pieces (modules) is the only way to manage it. After that division, the connected modules can be stashed from each other behind interfaces [25].

Module testing (MT) is performing tests that focus on testing slightly bigger functionalities of the system, compared with the smallest units that unit tests are focused on. However, often in literature MT is not separated from UT, but they are referred to be the same process, and modules are referred to be the smallest pieces [7]. In practice these processes are often separated, depending on the size of the project and if it is divided into modules, and of course if the separating is preferred by the organization.

When testing a module that has multiple possible use cases, different scenarios or many different configurations it can be used with, the optimal situation is that every possible configuration and situation get tested. This means multiple tests written for the module under testing. Tests created for MT focus on the validity of some functionality that often executes more than one function to produce the desired outcome. For

example, some test can verify whether user data forwarding works correctly in the eCPRI module. When creating a module divided embedded system, it provides useful information to test the modules with a suitable SoC or with the actual product HW, if it is already available. If the development is done in the host environment, the tests are convenient to run in there as well. Utilizing host testing together with product HW testing is called dual-targeting [26], which can be useful in separating the SW faults from the HW faults.

Due to the increasing of the functionalities belonging to a test in MT compared with UT, the complexity of the tests and testing process usually increases as well. Before running the tests, there often are many things needing to be setup correctly for the tests to work. Often there are different parameters that need to be set to correct states and multiple functions to be run for the test to be able to pass. These can be for example generating data which is going to be forwarded or initializing memory in order to make the test function correctly and test the right things with correct settings.

Setup function can be defined to handle function calls which initialize the situation for the test cases. The counterpart to setup function is teardown function, whose job is to set everything as they were before the test was run. These functions can be defined for each test group separately, to be suitable for the tests belonging to the groups. Setup function gets executed before every test in that test group, and teardown gets executed after a test has finished [27]. If there are differences for the needed parameters for some test in a test group, they can be set correctly in the test itself. If the test alters some global variable for example, the variable should be set back at the state it had before the test altered it. Otherwise, the test becomes state-polluting, which can lead to odd behaviour in later stages. It is also the job of teardown function to free the allocated memory, buffers or other resources that were reserved for the use of the test cases. If it fails to do so, a resource leak has occurred.

As these modules are part of a bigger entirety, they need to have interfaces to other modules they interact with. When the testing process is in MT phase, the other modules needed for interaction can be simulated, so the module's interfaces and integration to other modules can be tested. After the MT phase is completed, the integration of these independent but connected modules is tested in the next test phase, integration testing, as illustrated in Fig. 4 earlier.

4. FLAKY TESTS

Testing the software should always return the same outcome, when running the tests on the same code base, without changes. This should also be the case if the code stays the same, but the order of the tests is changed [28]. The outcome should always be the same. However, sometimes some test may fail, even if nothing has changed. These types of tests, that pass or fail in a nondeterministic manner, are often called flaky tests [5, 6, 29].

Test flakiness is seen as a severe problem with a lot of different kind of affects. Rerunning test cases due to their nondeterministic output by the developers or by the CI pipeline causes delays for developers which might end up even delaying the whole project they are working on. Flakiness also increases the usage of resources, which includes time and money in the top of the computational resources. In fact, from all the computational resource usage in Google, the part consisting from rerunning flaky test cases have been reported being as high as 16 % [30]. However, losing developers' time and trust towards the outcome of the tests is seen as a bigger issue than the increased usage of computational resources due to rerunning test cases [31].

There are multiple reasons why test cases can be flaky. The biggest reasons are order dependencies, asynchronous wait, concurrency and resource leaks [32, 33]. Which of these reasons is the biggest depends on the used coding languages, and of course there can be differences due to working habits and preferences between different companies, teams or even between developers.

4.1. Order-Dependent Tests

According to empirical analyses made of flaky tests, order-dependent flaky tests are one of the most common kind of flaky tests [32, 33]. Every single test should be independent, they should be able to run isolated and in any possible order, and pass every time when ran on the same code base [28]. However, in practise it is not that simple in large software projects. When tests use shared memory and global variables, flags and configurations, some tests might end up altering something from the shared pool and not restore it back to its original value. Tests are considered as polluting if they leave something changed from the shared pool after they have finished [5]. When another test is developed, it checks the value of that shared variable altered by some earlier test without setting it itself, and it passes because the variable is suitable for it. If the order of these tests gets changed, it may not pass anymore, because the variable might be on its initialized value instead of the modified one, because the state-polluting test might not have been executed yet.

Accidentally generating dependencies between test cases can happen rather easily in large software projects. When writing tests for a project that has multiple teams with many people using the same code repository, and perhaps the tests are always run in the same order, even one incomplete teardown function can cause dependencies between multiple test cases. This situation is illustrated in Fig. 6, where Test 2 alters shared variable in its setup function. However, the variable is not restored back to its original value in the test's teardown function. Test 3 also uses the variable, but it is already set to be suitable for it by the previous test, so it does not set it again. However, test 3

checks the value in the test and the check passes. A dependency has occurred between Test 2 and Test 3.

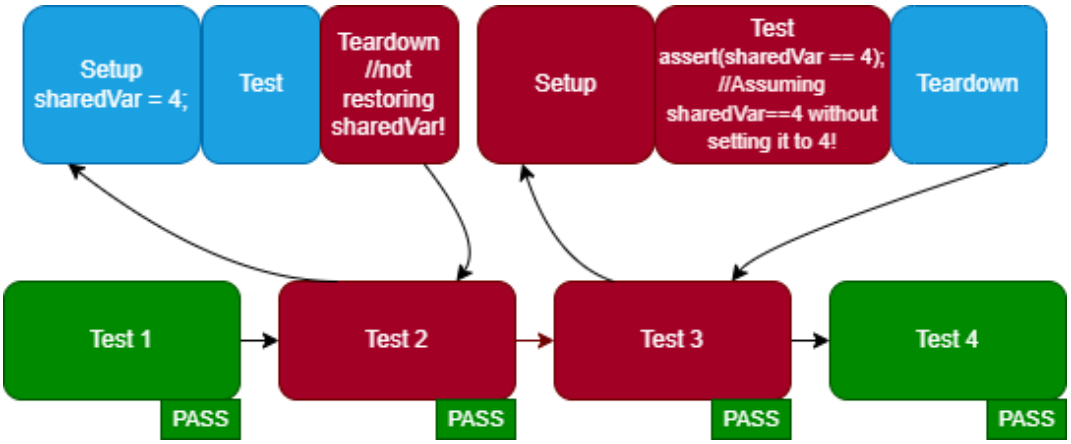

Figure 6. Illustration of an order dependency between Test 2 and Test 3.

If these tests are run in a shuffled order, so that Test 3 gets executed before Test 2, Test 3 will fail on the assertion of that shared variable. This is illustrated on Fig. 7. Always running the tests in the same order might hide the dependencies between tests. If the test order or some of the tests itself are modified or disabled at later point, the dependencies might come up and the modifications might break the test job. This would then require some effort to fix the test cases so that they are able to pass again. Depending on the scope of the software under testing and its complexity, the dependency might be hard to locate and fixing the job might take a lot of resources from the development team.

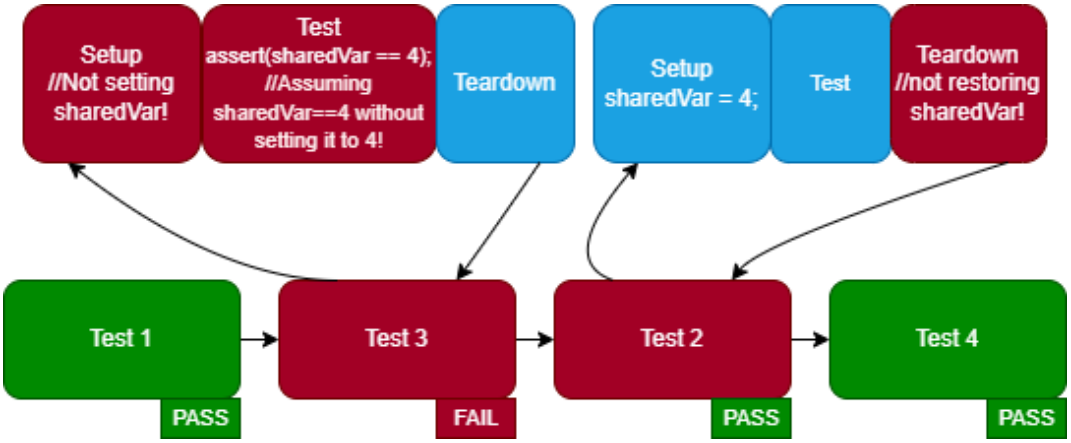

Figure 7. Tests are run in shuffled order, so that Test 3 gets executed before Test 2. Test 3 fails to an assertion because Test 3 is dependent on Test 2. Due to the shuffled test order, the shared variable is not correct for Test 2.

4.2. Resource Leaks

Resource or memory leaks are common reasons for test flakiness. Resource leak happens when the resources of the application are not managed properly [32]. Leaks can happen due to a failure in acquiring or releasing the needed resource, but also if acquiring or releasing resources unnecessarily. If a test allocates some memory, a buffer, or creates transmission packets for it to use for example, but does not free it or destroy the packets after the testing is done, a leak has occurred. Failing to release acquired memory or resources might be due to failing to meet some critical application dependent requirements before releasing the resource, or just simply due to a missing release command in the test group's teardown function for example. Tools that manage and observe the memory usage of the application, such as Valgrind [34], should be able to find these leaks, but that is not always the case. For example, if Valgrind assumes that the reserved memory is needed until the end of the execution of the application, it might not announce it as a memory leak. An example of a resource leak is illustrated in Fig. 8.

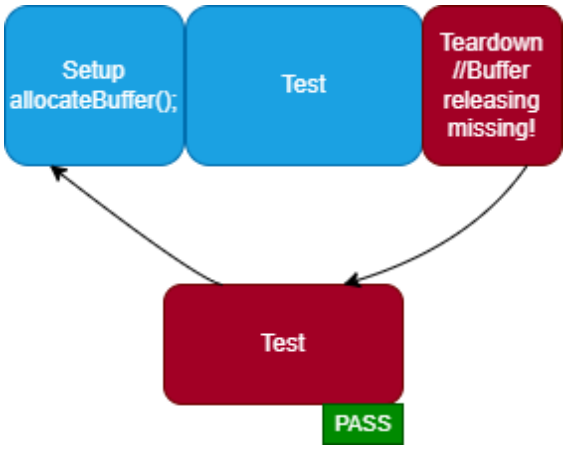

Figure 8. An example of a resource leak. The buffer is allocated in the test's setup function, but it is not released in its teardown function. The test will still pass.

If resource leaks go undetected, they might end up creating dependencies between tests. A possible case of resource leak leading to an order dependency is illustrated in Fig. 9, where Test 2 allocates a buffer and does not release it, and Test 3 does not allocate that buffer, but releases it instead. If the buffer is always locked when reserved by another test, or it is full after Test 2 has finished and it cannot be written over, the unreleased buffer would most likely be caught in the writing phase for Test 3 at the latest. If Test 3 can use it however, the buffer allocation might be missing from its setup function, because the test works without it. The developer might initially have called the allocation function for the buffer on Test 3 setup function, but later on taken it out due to a crash occurring from a double allocation of that buffer. Correct fix for this would have been releasing the buffer in Test 2 teardown function and reserving it for Test 3. Depending on how the buffer is allocated and managed, Test 3 might be able to use it, but it might also fail to do so, which can result as a flaky test case.

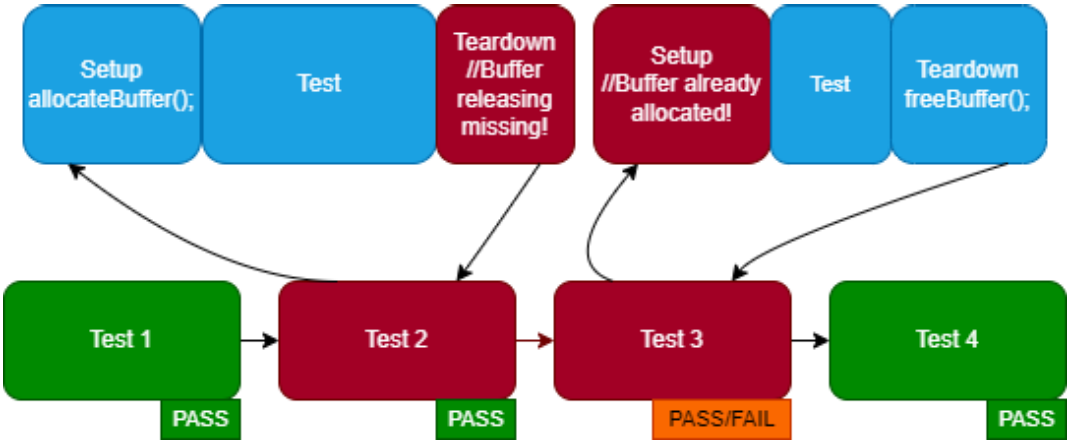

Figure 9. Resource leak leading to order dependency.

4.3. Preventing Flaky Tests

Preventing flaky test cases is not always so simple, especially when working in a large project with a big number of developers with different backgrounds, languages, education and time zones. One could easily assume that if the development of the tests and the application is done carefully and by following the principles of clean coding, the tests will not be flaky. That can be true to some point and in some situations, but when there are multiple different developers, teams, platforms and projects using and changing the same code base, the issue is not that simple anymore. When adding the pressure coming from the timeline of the project and from the competitors on top of that, some imperfections might get through the code reviews, especially when the tests checking the proposed changes are passing at the time when they are being introduced to the master repository.

Even though good coding principles and extensive regression testing makes an effort to prevent developing bad code which is prone to non-deterministic behaviour in tests, that is not always enough on its own to fully ensure that flaky tests will not be introduced. There are not any single magic ways to fully prevent those from occurring. Designing test code to avoid flakiness is described being an important challenge to face and the research of it requires further investigation [35]. The absence of literature on how to prevent developing flaky tests tells that there is a need for deeper understanding and common principles on the subject. However, there is literature and tools that focus on detecting and fixing these tests.

Many of the tools created to automatically detect flaky tests are designed for some specific types of flakiness and for specific platforms. Examples of these tools for Java are PRADET focusing on detecting dependencies [6] and FlakeFlagger predicting tests to be flaky based on identifying flaky parts in them [36]. Another example is DeFlaker, which compares the coverage of the change to the coverage of the failing test case to see if the change caused the failure [37]. For Android applications, there is SHAKER which adds noise to the execution environment [38]. Other examples are Multi-factor approach MDFlaker [39] for Python, and an algorithm named Flakiness Debugger investigating divergences between runs, which can be used with C++ and

Java [40]. Such things have also been proposed as preventing methods like tools utilizing machine learning to spot patterns in tests that are marks of flakiness, which could help developers to spot tests that have a high probability of becoming flaky in the future [35].

However, even if the existing tools often introduce some limitations that make them not usable in some specific projects, or if the introduction of some external tool is not desired, the ideology behind them can perhaps be applied, depending on the project. This of course requires a lot of effort from the developers to implement these kinds of tools basically from the scratch, if the existing ones are not open-sourced.

Some of those tools and methods could possibly be used as preventive methods in order to decrease the chance of introducing flaky tests to the project. In addition to clean coding and strict following of good coding principles, one could try to avoid order dependencies by running the test cases in shuffled order, instead of in the same order every time. This would, if not prevent the order dependencies completely, at least bring them up earlier after they are mistakenly introduced. Also, as test rerunning is a part of many of the existing tools and considered as one the most common approaches to detect them [39], the tests could be repeated some amount of times periodically to possibly bring up flaky test cases. And, if suitable for the project, the automatic detectors could be set to run right at the start of the project, so when some bad-quality test code or some marks of flakiness are introduced, they could be caught early on, assuming the proper functionality of the chosen tool. Most of these preventative methods requires an increase of resource usage. However, in the long run they might end up being beneficial and saving resources and time if they are able to prevent at least some of flaky tests which would otherwise be introduced.

5. SITUATION AND BACKGROUND

5.1. Goal and Motivation

This work focuses on creating tools for helping in locating, debugging and fixing unstable test cases in the eCPRI module under development in 5G L1 FH SW area. Since most of the existing tools are not designed for C++ and do not necessarily suit the project either, or it is not desired to take some third-party tool in use, an in-house implementation is required. The goal is to create tools for easing the process of finding and fixing the reasons why these test cases fail in a nondeterministic manner and to increase the reproducibility of the test failures. Fixing the flakiness of the unstable test cases would result in better test stability, faster CI pipeline and development process, decreased resource usage, and savings on the costs of developing this module. While developers do not need to use so much effort trying to find and fix unstable test cases, their time and effort can be addressed to actually develop the module forwards, instead of use a big portion of their time trying to locate bugs that appear now and then, and that are often hard to reproduce.

When tests fail frequently in CI pipeline used by also other developers who are focusing on other modules or projects, their work gets delayed as well, due to the need of rerunning tests if they have failed. This leads to a situation that their proposed changes to the code cannot be integrated as fast to the master repository as they could if the tests were stable. This problem can partly be tackled by configuring the CI pipeline to treat these tests as non-blocking or to disable those tests from running on the platform they are unstable on. However, that could expose the module for bugs and misbehaviour, since the testing of it would not cover the module as widely as it should. The desired situation would be to have every test enabled and them to run successfully every time when ran on the same version of the code. That would prevent the introduction of new bugs, keep the CI pipeline fast, and reduce the costs.

5.2. Environment

The testing phases used with RAN systems follows the same principles as almost any other embedded system that is divided into multiple modules by SW and/or HW level. Generalized testing phases for RAN systems are illustrated in Fig. 10. The first step is UT, which is started at the very beginning of the development. UT is followed by MT, where the system under testing (SUT) is a single module. It is also beneficial to test the interfaces the module has, and it is usually done by simulating the connected modules other side of the interface.

After MT, the next step is integration testing. In integration testing, the SUT is a set of connected modules, and the target is to see if they work correctly together. For RAN, the integration testing can be done by layers, so the SUT can be L1 for example. In this case, the L1's interfaces could be tested through a simulated RF block and L2. When the integration testing is finished, the next step is to test the new SW releases on the product HW. This phase is called Quick Test. Entity Testing is the following step, where the completed SW is tested with every HW module there is. External equipment is simulated in this phase, such as UEs (User Equipment) for a

RAN system for example. The final step is System Testing, where the SUT is the entire system where nothing is simulated, and the goal is to ensure that everything is working properly. Everything is tested as they will be when the product is launched for commercial use and provided for customers.

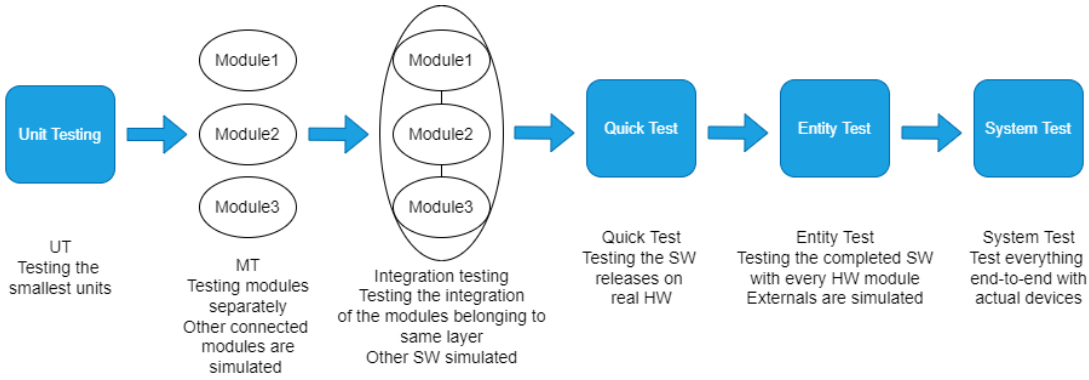

Figure 10. Testing phases for RAN system.

The development of this product is currently in the MT phase. Currently, the stability issues in testing are seen with the MTs used with the new eCPRI module under development, so this thesis focuses on the tests developed for that. However, the tools created in this thesis are designed to be general, so that they can be used with other than MTs as well, for example with UTs or later on with integration testing. To be able to use all the tools with different scenarios, the tests need to be using the CppUTest framework. If the framework is not in use in the integration testing phase, the tools need to be modified to fit for that, since they are using the CppUTest command line switches. The CppUTest framework and its command line switches get presented in Chapter 5.3.

While developing embedded software, the development and testing of the module under development is usually done in a host environment at first. This is because the actual processor is usually tuned to be as energy-saving as possible, so any extra functionalities such as support for development tools and high computational power are often ruled out, so using a host environment is more convenient [41]. Testing with the product HW should never be ruled out though, since it is the actual HW that the software will be operated on. It is more efficient to develop the software and the tests for it in a host environment, and then use cross-compilers and linkers to generate the code to be tested with the product HW. This reduces the required amount of the product HWs existing early in the development phase, since every developer does not need to have the actual device constantly to be able to progress in the development.

Locating the flaky tests using the tools created is done by running the test on host environment. Currently, testing with the host includes the biggest number of tests, is the fastest and does not require as many resources, due to the lack of a need for any product HW or even simulated ones. Testing with the product HW would reserve the board, denying any usage from other users with that HW. This would cause delays for other developers who are waiting to test and develop the module with it, especially when extensive testing is performed. Testing with tests running on host uses only processing resources from the server used to run the tests. When extensive and

automated testing is performed overnight, the negative influence on other developers' work is minimal. The tests are mostly the same as the tests which get executed in the runs that are performed with the product HW, so fixing instabilities, memory leaks or dependencies with host testing should also provide better stability for product HW runs.

Testing and developing an embedded software for a product that is divided into modules for the ability to actually develop and handle it, some processing and other functionalities might come from outside of the module being developed. If those external sources provide something that causes instabilities to the application or to the tests, the locating and fixing the issues might be very hard, since there might not be any visibility to inside of those external sources.

Since there are already tools for monitoring memory leaks, and because there are many indicators that are suggesting that order dependencies exist, the work of this thesis focuses on revealing and fixing dependencies between tests and test groups, which is a big reason for nondeterministic behaviour in test cases.

5.3. CppUTest Framework

The MTs and UTs used in this project are built using CppUTest. CppUTest is an open source, xUnit test framework designed and based on C/C++ languages, and it is often used with embedded systems [42]. CppUTest provides multiple different helpful options for the management of the tests, such as test macros which help to create test groups, ignore single tests for everything or for specific builds, base class inheritance and many useful assertions the tests can use. The framework also offers a possibility to integrate GoogleTest, which is Google's framework for testing and mocking [43], together with CppUTest into the same binary, as well as to mock with CppUMock and GMock together [27].

The framework also provides many command-line switches, from which the most useful ones for this project are presented in Table 1 [27]. These command line switches provide useful information, and they can be used to execute the tests in a different way with rather low effort. Switches *-lg* and *-ln* provide lists of the test groups or test cases that get executed with the given test command. This information is used with some of the tools introduced in this thesis. Switch *-r* repeats the tests under the same start up a given number of times, which can be useful to detect resource leaks. Shuffling the test order can be done with the switch *-s*, and it can be given some seed value what is used for shuffling, or if it is left blank, time-based seed is used each time a test run starts, resulting in constantly changing seed. Switch *-sg* runs only those test groups that match the given string and switch *-sn* compares the string to test names, running only the tests that match the string. These can be useful to investigate whether the test groups or test cases are independent. Switches *-xg* and *-xn* excludes matching test groups or single tests, which can be used to detect dependencies between test cases for example.

Table 1. Some of the CppUTest command line switches and their functionalities

Switch	Functionality
-lg	Prints a list containing every group name, separated by spaces
-ln	Prints a list containing every test name in a form of test.group, space separated
-r #	Runs tests repeatedly # times in one startup. Twice if # is not specified
-s *seed*	Shuffles the test order with given seed. Uses time based seed if not specified
-sg *group*	Runs only tests which group is an exact match to the string *group*
-sn *name*	Runs only tests which name is an exact match to the string *name*
-xg *group*	Excludes tests which group contains the string *group*
-xn *name*	Excludes tests which name contains the string *name*

5.4. Compilers

There are two different compilers supported for the product under development in host test environment, which are Clang and GCC. Compiling time for GCC is much higher than for Clang, and therefore developers often prefer to use Clang as their primary compiler. Running a round of these tests with GCC, without the need for building the target first, takes approximately five minutes depending on the current load of the server and whether a completely new build is required for the run. More instabilities are seen on the Clang compiler, however. GCC executes the tests in an opposite order compared with Clang, and some tests are currently flagged only to be executed for only one of these compilers.

Due to the fact that GCC and Clang executes given test groups in opposite order due to an opposite direction used when linking the files, there are currently different definitions for each compiler on what to run and in what order. This is currently needed because if using a common one for these compilers, either one fails. This makes it clear that the execution order of the tests matters, which indicates that there are dependencies between test cases. Having two different definitions for what tests to run is not an optimal situation. When introducing new test files, the developer can mistakenly add the test file only to be executed on one of the compilers. The same thing can happen when taking outdated test files out of being executed. Maintaining only one definition of what tests to build and run for a given target would be much more sustainable.

5.5. Initial Stability

The statistics from the initial situation with the Clang compiler can be examined from Table 2. At the beginning of this work, running all tests which were enabled for host test runs (194 tests) with the Clang compiler with a fresh start after previous completed test run, the test runs succeeded (every test passed) 165 times out of 200 runs, leading to a success percent of 82.5%. This means that 17.5% percent of test runs on host with

this compiler failed. When enabling all the existing test cases that were disabled but are capable of passing taking into account the state of the product code, the corresponding numbers are 222 tests on a run and 116 runs passing from 200, reducing the success percent to be 58%. When shuffling the test order to acquire information about possible order dependencies, the success percent reduces to zero in 200 rounds. One test group is excluded from the shuffle runs from the initial stability runs as well as the result runs presented at the end of this thesis, as it cannot be shuffled at this point due to external dependencies. With the absence of successful test runs when shuffling the order, it can be confirmed that there indeed are order dependencies between test cases, and that they are quite a big problem.

Table 2. Statistics from 200 test runs in initial situation for the Clang compiler

Configuration	Tests in a run	Passed runs	Success %
Clang	194	165	82.5%
Clang All Enabled	222	116	58%
Clang Shuffled	185	0	0%

The same statistics for the GCC compiler are 192 tests passing on 200 out of 200 runs, leading to a perfect success percent of 100%. All tests (220) enabled, 200 passed from 200 runs again, so enabling the disabled test cases does not have an effect on the stability when using GCC as the compiler. However, when shuffling the order with GCC, the success percent goes to zero as well, despite the fact that the initial success percent was much higher with GCC than it was with Clang, when running the tests in a normal order. The statistics for the GCC compiler in the initial situation can be seen from Table 3.

Table 3. Statistics from 200 test runs in initial situation for the GCC compiler

Configuration	Tests in a run	Passed runs	Success %
GCC	192	200	100%
GCC All Enabled	220	200	100%
GCC Shuffled	183	0	0%

6. TOOLS

To help to bring up the flakiness and testing different subjects that can be factors of the stability of the test cases, a set of tools needed to be created. These tools, the functions in them and how to use the tools are gone through in this chapter. The tools can also be found in the appendices of this thesis.

6.1. Test Repeater

Collecting logs from failed runs to acquire important information about the possible reasons for the failures can be difficult due to the failures occurring rather rarely and in a nondeterministic manner. If some test case fails with a probability of 5 percent, which means once in every 20 runs in average, manually hunting down the failure can take a lot of time, especially since the reasons for that failure are still unknown. The need of starting the test run again manually after every successful test run requires constant awareness and presence from the developer trying to find the failing run to get logs from it. This consumes the developer's time and can distract them from focusing on other subjects at the same time.

Using the command line switch provided by CppUTest for repeating the tests does not start the application again after the tests have run, but instead it keeps the application running, only starting the tests again. This leads to a different situation than what happens in a normal, single run, which is how they are executed in the CI pipeline. Repeating the tests under the same start up can be beneficial for locating possible reasons for failures as well, but those possible failures might be due to completely different reasons than the issue taken under investigation. Automatically running the test cases repeatedly with a clean start up every time, as they are run in the CI pipeline, decreases the amount of effort the developers need to put in to find a failed run so that they can acquire logs and information from it.

For developers being able to leave tests running automatically for a desired number of times with a clean start up for every single run, a Python script was developed. Previously, to achieve an automated and separated test running for several runs, the developers needed to create reviews about their changes, so the job validator of the CI pipeline could be used. It is not convenient, however, to constantly push small changes to the review platform only for the ability to validate them or to find a randomly occurring failure they are trying to debug. With the introduction of this script, the developers could start an automated and separated test running with one command directly from their developing environment, without the need for pushing their changes to the reviewing platform. The script can for example be left running overnight without the need for being constantly present and starting the tests again. This way the developer can get the log files ready for the next working day, with only the effort of starting the script for the desired amount of test runs at the end of the previous day. The required time for finishing the runs depends on the amount of test cases and runs of course. Most important functions are introduced and the whole code can be investigated from Appendix 1.

6.1.1. How to Use

Table 4. Arguments the Test Repeater -script takes in

Argument	Explanation	Example
-s, --single	Command to run once, optional	-s 'BUILD COMMAND'
-t, --test	Command to repeat, mandatory	-t 'TEST COMMAND'
-r, --repeat	How many times to repeat, optional	-r 20
-l, --log	Save 'all', 'failed' or 'none' logs	-l 'failed'

The Test Repeater script is written with Python, and it executes the tests by using the command line to input commands the same way the developers do. The arguments the script takes in are presented in Table 4. User can define two different commands to execute, one of which, given with argument *-s*, gets executed only once. This is useful for example when the developer has made some changes to the code, and the build command needs to be executed before the test command, for the changes to be taken into account. The other command to be executed, given with an argument *-t*, is the test command that gets repeated a defined number of times. Argument *-r* is used to define the value how many times the given test command is repeated with a clean start up. The last argument -l defines which logs are saved, based on the outcome of the test run. The command to be repeated can include the command line switches that can be given to tests running with the CppUTest framework, which were introduced earlier in Table 1. If the tests are run with the shuffle flag on, the script searches the seed from the log of a failed run, saves it and prints it for the developer after all the test runs are executed. This helps the user to rerun the tests with the seeds the possible failures existed.

The script is executed by starting the script from the command line just like any other Python script with suitable arguments given. Argument handling and the instructions for them are done with a Python package named *argparse* [44]. Before the script starts going through the main function, the packages needed to run the script are imported, some objects and variables are initialized, starting time gets saved and the path for results gets defined based on the starting time and pre-defined directory. These can be seen from Fig. 11.

```
#!/usr/bin/env python3

# This script runs all given tests repeatedly
# for given number of times, with a new start up.
# Made to monitor unstability of tests and to
# locate flakiness.

import argparse
import logging
import os
import shutil
import subprocess
import sys
import time

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("Test Repeater")
parser = argparse.ArgumentParser()

starttime = time.time()
RESULTS_DIR = 'out/logs/repeat_results/' + str(int(starttime)) + '/'
numOfFailedRuns = 0
numOfPassedRuns = 0
failedSeeds = ''
```

Figure 11. Imports, definitions and object initialization that gets executed before the main -function.

6.1.2. ParseArgs -Function

This function handles the argument parsing by using the "argparse' package. An ArgumentParser object has to be created, which is done when starting up the script. After that, it can be given arguments what the script should be expecting. The developer can define the names, shortcuts and types of the arguments, and the ArgumentParser object will take care that the user of the script gives the arguments correctly. If not, help messages defined by the developer are print for the user. The same help messages can be acquired by calling the script with argument -h, which is automatically created by the parser. The function returns the parsed arguments for the user.

```
def parseArgs():
    parser.add_argument('--single', '-s',
                        help="Cmd to ran once", type=str)
    parser.add_argument('--test', '-t',
                        help="Cmd to repeat, ran X times", type=str)
    parser.add_argument('--repeat', '-r',
                        help="How many times to repeat", type=int, default=1)
    parser.add_argument('--log', '-l',
                        help="Save 'all', 'failed' or 'none' logs",
                        type=str, default='all')
    return parser.parse_args()
```

Figure 12. parseArgs -function.

6.1.3. Shell -Function

Shell -function defined for this script takes in two arguments: *cmd* and *check*. The script executes command given with the argument *cmd* in the command line the same way the user manually executes them, with the exception that the user can define with the *check* argument whether a crash occurs or not, if the command that gets executed does not return zero. If *check* is defined to be *True*, which it is defaulting to, the script raises an error if the return value is not zero, and the script gets shut down. This is used for example with the command that is given with argument -s, often used with a build command. The starting of running the tests is prevented with this if the build command fails. Using *check=False* enables the repeating of the test runs also in the case if one of those runs reports a failure.

The shell -function also prints information for the user, such as what command is being executed currently, and also it prints the ignored failure if *check=False* and an error occurs. The shell -function returns the return code of the result as an output.

```
def shell(cmd, check=True):
    logger.info('shell: %s', cmd)
    result = subprocess.run(cmd, check=check, shell=True,
                            stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
                            executable=shutil.which('bash'))
    if not check and result.returncode != 0:
        logger.info('ignoring failure: %s', result)
    subprocess.run("echo -ne '\n'", check=check,
                   shell=True, executable=shutil.which('bash'))
    return result.returncode
```

Figure 13. shell -function.

6.1.4. GetSeed -Function

This function gets called when a test run fails and the command line switch *-s* for shuffling the order of the test cases is used. The seed used for shuffling gets fetched from the test execution log file. For fetching the seed, shell command *grep* is used with suitable regular expression search, so only the actual seed gets picked out from the log file. The function also includes an assertion that checks that the seed was found from the log file. As this function gets called only if shuffling is used, so the seed should always be found from the log file. The function returns the seed as a cleaned string for easy saving later on in the script.

```
def getSeed():
    cmd = "grep '(?<=seed: )[^ ]*' -oP" + RESULTS_DIR + "templog.txt"
    result = subprocess.run(cmd, shell=True,
                            stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
                            executable=shutil.which('bash'))
    assert result.returncode == 0, "Seed not found!"
    return str(result.stdout).strip("b'\\n")
```

Figure 14. getSeed -function.

6.1.5. LoopTests -Function

This function is used for looping the tests a desired number of times. The function takes in arguments *cmd*, which holds the command to be looped, *repeatTimes* which defines the amount of how many times the command gets looped, *dir* which defines the path to the directory the logs get saved, and *log* which defines which of the logs gets saved, based on the argument given when starting the script, defaulting to 'all'.

In addition to looping the tests, the function creates directories under the path given with *dir* for the log files, based on the value of the argument *log*, by using shell command *mkdir*. The function also updates the global variables of passed and failed runs, and based on the outcome of the run and the value of the *log* -argument, moves the log file under the correct folder. If a test run fails, the command gets checked whether it contains the command line switch *-s*. If so, the seed used for shuffling gets fetched by calling the *getSeed* -function, and it gets saved to a string containing possible previous seeds that were used in some failing run occurring under the current usage of the script.

```
def loopTests(cmd, repeatTimes, dir=RESULTS_DIR, log='all'):
    global numOfFailedRuns, numOfPassedRuns, failedSeeds
    tempLog = dir + "templog.txt "

    shell("mkdir -p " + dir)
    for i in range(repeatTimes):
        statusMsg= "Round " + str(i+1) + "/" + str(repeatTimes)
        logger.info(statusMsg)
        result = shell(cmd + " > " + tempLog, check=False)
        if result == 0:
            numOfPassedRuns += 1
            if log == 'all':
                shell("mkdir -p " + dir + "passed/")
                saveLog = "mv " + tempLog + dir + "passed/log"
                shell(saveLog + str(i) + ".txt")
            else:
                shell("rm " + tempLog)
        else:
            numOfFailedRuns += 1
            if '-s ' in cmd:
                seed = getSeed()
                failedSeeds = failedSeeds + seed + ' '
            if log != 'none':
                shell("mkdir -p " + dir + "failed/")
                saveLog = "mv " + tempLog + dir + "failed/log"
                shell(saveLog + str(i) + ".txt")
```

Figure 15. loopTests -function.

6.1.6. SaveResults -Function

The saveResults -function saves the outcome of the test runs into a file under the given directory. It takes in arguments *cmd* containing the test command, *dir* for the output directory and *log* to see if the results get saved or only printed. The function forms a string containing information on how many test runs were passed and failed based on the global variables, as well as the corresponding percent for them. Execution time also gets printed. The function checks if the global variable *failedSeeds* contains some seed, and if so, the seeds used in failed shuffled runs gets added to the result string in the order they existed. The function also provides the command used with the first failing shuffled test run, so the user can easily repeat the command that brought up the failure. The combined result string gets printed to the console, and then saved as a file to the given directory as a file named *results.txt*, if logging is not set to *none*.

```
def saveResults(cmd, dir, log):
    runs = numOfPassedRuns + numOfFailedRuns
    if runs == 0:
        logger.info("\nNo tests executed!")
        return
    resultStr = ("\nPASSED: " + str(numOfPassedRuns) + " TIMES, " +
                 str(round((numOfPassedRuns/runs)*100, 2)) + "%\n"
                 "FAILED: " + str(numOfFailedRuns) + " TIMES, " +
                 str(round((numOfFailedRuns/runs)*100, 2)) + "%\n"
                 "execution time, s: " + str(round((time.time()-starttime), 1)))
    if failedSeeds != '':
        resultStr = (resultStr + '\nShuffle seeds in failing order: ' +
                     failedSeeds)
        helpCmd = cmd.split("-s ")
        resultStr = (resultStr + '\n\nTest this manually with command:\n' +
                     helpCmd[0] + '-s ' + failedSeeds.split(' ')[0] + ' ' +
                     helpCmd[1])
    logger.info(resultStr + '\n')
    if log != 'none':
        with open(dir + 'results.txt', 'w') as resultFile:
            resultFile.write(resultStr)
```

Figure 16. saveResults -function.

6.1.7. Main -Function

Main function defines on which order and how the introduced functions get called. Arguments are fetched by calling parseArgs -function, and if build command – or some other command that needs to be executed once before the looping starts – is given with the *-s* argument, the given command gets ran and the log from that is directed to a file named *buildlog.txt*. Next step is to try to call the loopTests -function with the arguments the user gave. *TypeError* is expected, as it pops up if the user does not give the required arguments for the script. If the error is raised, printInstructions -function gets called to provide help and example command for the user. If the error is not raised, which means the user provided correct arguments, the test looping gets started. Main

-function returns the global variable *numOfFailedRuns*, which functions as the exit code the script provides. If every test run passes, the script returns value zero, which is a sign of a successful execution of the script. If some other value is returned, it means that many test runs failed.

```
def main():
    args = parseArgs()
    shell("mkdir -p " + RESULTS_DIR)
    if args.single != None:
        shell(args.single + " > " + RESULTS_DIR + "buildlog.txt")
    try:
        cmd = args.test
        if "-s'" in cmd:
            cmd = cmd.replace("-s'", "-s '")
        loopTests(cmd, args.repeat, log=args.log)
        saveResults(cmd, RESULTS_DIR, args.log)
        if args.log == 'none':
            removeDir(RESULTS_DIR)
    except TypeError:
        printInstructions()
    return numOfFailedRuns

if __name__ == '__main__':
    sys.exit(main())
```

Figure 17. main -function.

6.2. Single Test Repeater

Single Test Repeater is rather similar to the Test Repeater, and it utilizes few things directly from it. Functions loopTests, parseArgs, removeDir, shell, saveResults and the parser -object are imported from it for this script to use. This reduces the need for refactoring when there are changes made to the code. The function also has nearly similar printInstructions -function as the Test Repeater has, only the example command is different, and a note is added which clarifies that the *-sn* command line switch must be used with this tool. The Single Test Repeater can be used with the same arguments as the Test Repeater. The whole tool can be investigated from Appendix 2.

6.2.1. GetTestnamesToFile -Function

To be able to automatize the separated single test running, the names of the test cases must be acquired. This function utilizes the CppUTest command line switch *-ln* to acquire the test names. The shell function is utilized to run the same shell command that is given to be executed, but the required argument *-sn* gets replaced by *-ln*. This produces a list of the test groups and names, which is directed to a log file named *testlog.txt*. After the creation of that file, the shell -function is again used to run suitable *grep* command with regular expressions combined with commands *sed* and *tr*, which forms a list of test names and test groups that are executed in the given test binary, in

a format of test.group, each of which having their own line. This list of tests is then directed to *TESTS_FILE*, which is defined in the start up phase of the script to be a file named *tests.txt*, locating in the result directory.

```
def getTestnamesToFile(cmd):
    pathToFile = RESULTS_DIR + "testList.txt"
    shell(cmd.replace('-sn', '-ln') + " > " + pathToFile)
    shell("grep '^[^0-9]' " + pathToFile + " | grep ' Test' | " +
          "sed 's/[[:digit:]]\+:.*//' | tr ' ' '\\n' > " + TESTS_FILE)
```

Figure 18. getTestnamesToFile -function.

6.2.2. Main -Function

The main -function of the Single Test Repeater is based on the Test Repeater, so it is also rather similar to that. The function starts by fetching the arguments, which is done by the parseArgs -function imported from Test Repeater, and by checking the existence of the single execution argument. Next, the test argument is checked that if it includes the *-sn* command line switch. If it is missing, a TypeError is raised, and the printInstructions -function gets called. If the command line switch is used, the getTestnamesToFile gets called to acquire the test names. The following step is to split the test command from the location of the *-sn* switch, since that is the place where the test names need to be injected. To acquire the name of the tests, the *TESTS_FILE* gets opened and lines get taken into use one by one. The test name gets separated from the test group, and finally the loopTests -function gets called for each test, one by one with a new start up.

```
def main():
    args = parseArgs()
    shell("mkdir -p " + RESULTS_DIR)
    if args.single != None:
        shell(args.single + " > " + RESULTS_DIR + "buildlog.txt")
    try:
        if '-sn' not in args.test:
            raise TypeError
        getTestnamesToFile(args.test)
        cmd = args.test.split('-sn')
        with open(TESTS_FILE, 'r') as tests:
            for test in tests:
                test = test.replace('\n', '').split('.')
                testCmd = cmd[0] + ' -sn ' + test[1] + cmd[1]
                testDir = RESULTS_DIR + test[1] + '/'
                loopTests(testCmd, args.repeat, testDir, args.log)
                removeDir(testDir, onlyIfEmpty=True)
        saveResults(args.test, RESULTS_DIR, args.log)
        if args.log == 'none':
            removeDir(RESULTS_DIR)
    except TypeError:
        printInstructions()

if __name__ == '__main__':
    sys.exit(main())
```

Figure 19. main -function of the Single Test Repeater.

6.3. Single Test Group Repeater

This tool is very similar to the Single Test Repeater. Command line switches *-sn* and *ln* are replaced with switches *-sg* and *-lg*, to handle test groups instead of test cases. Test group looping in the main function requires less processing, since the command line switch *-lg* returns only the test groups. Otherwise, the tool is built basically the same way as the Single Test Repeater is. The tool can be examined from Appendix 3.

6.4. Discussion

All in all, three different but similar kinds of tools were built for investigating instabilities in test cases. All the tools are designed to be general, so that they can be used with tests of a different kind. However, to have every feature enabled, the tests must be built on CppUTest framework, since the tools utilize the framework's command line switches. With small alterations to the tools, they can be utilized with different ways to examine different kind of order dependencies. In addition to order dependency investigation, the Test Repeater -tool offers an automated test running, which frees the developer running the test cases to other duties. It also offers information about the stability level of the tests it is being used with, as well as centralized information about which are the test cases that keep failing.

7. METHODS AND RESULTS

This chapter presents how the tools created were used to locate and fix instabilities and order dependencies between tests or test groups, and what preventative methods were taken in use to try to block new dependencies occurring while the development of the module proceeds. All the methods presented below are executing tests with the Clang compiler, as it proved to be less stable due a to higher number of test failures with a normal test order. Since the objective is to discover reasons for flakiness in test cases and to reveal order dependencies, it is more beneficial to use the compiler which proved to be more unstable.

7.1. Utilization of Single Test Repeater and Single Test Group Repeater

7.1.1. Single Test and Test Group Execution

The Single Test Repeater was used to see if there are test cases which require, in order to be successful, the execution of some other test case beforehand. This method should catch the test cases that are written based on a polluted state coming from another test case. This was done by using the Single Test Repeater -script with basic settings, only using the CppUTest command line switch *-sn*, which is used for running only specific test cases and is required by the script to run the tests separately every time with a new start up.

The result of this investigation was that every test case was able to pass. This outcome proved that there are no test cases which have direct requirements of other test cases being executed before them, so the test cases are independent. This ruled out the possible scenario that some tests are accidentally written the way that they expect something to be as some other test changed them to be and left them that way, in a polluted state. In conclusion, the tests are setting the required values correctly either in their setup function or in their test bodies, or that they are relying on them to be as they are originally set by the application in its start up phase.

The same procedure was used for test groups by utilizing Single Test Group Repeater tool, only using command line switch *-sg*, which runs only specific test groups and is required by the script to run a single test group separately with a new start up every time. The outcome was the same as it was in the single test execution method. Every test group passed when running separately, so none of the test groups require some other test group to pollute the state before they are executed. This leads to a conclusion that the test groups are independent of other test groups.

7.1.2. Single Test and Test Group Exclusion

Single Test Repeater can be changed from running one test at a time to exclude one test at a time. This can be achieved by altering the script by changing the required command line switch from *-sn* to *-xn*, which excludes the test cases which name contains the given string. Utilizing the script with this slight modification provides more information about possible order dependencies or resource leaks in the tests. If

Test 1 leaves something changed, Test 2 does not check it but restores it back to the initial state, and Test 3 assumes that everything is in their initial state, the Test 3 starts to fail if Test 2 is excluded. This situation is illustrated in Fig. 20. Dependencies of this kind do not come up when running test cases or groups separately.

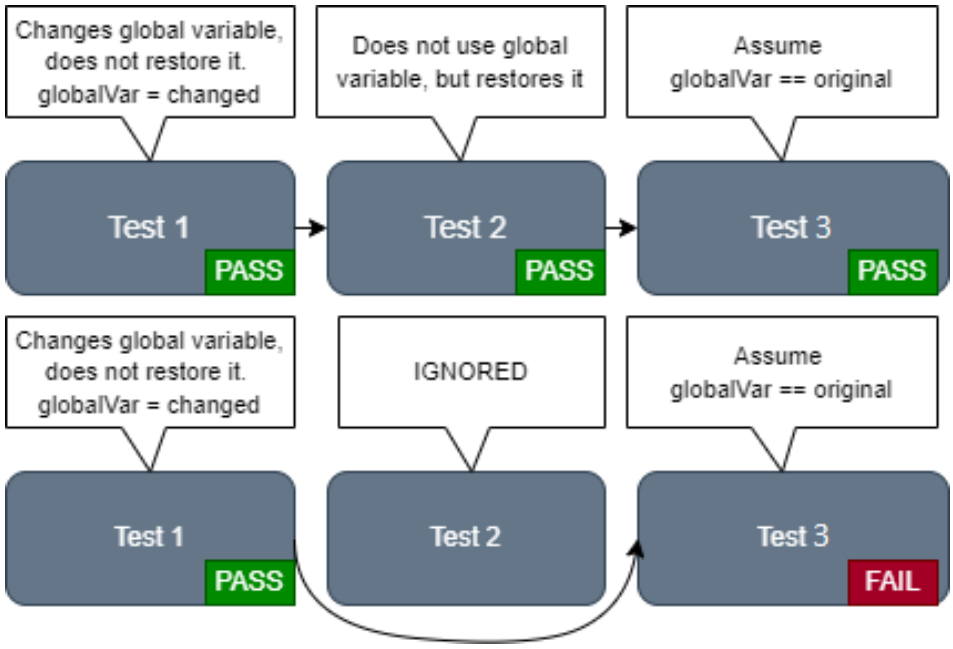

Figure 20. Illustration of an order dependency brought up by test exclusion.

However, this kind of testing is not so reasonable to perform very often. Excluding one test case at a time requires many demanding test runs, since there are as many runs as there are separate test cases, and in each of those runs there are all but one test case being executed. And this kind of test dependency does not happen so easily, as it is likely to be caught by checks when writing the test cases. Despite that, this method was used to check whether this kind of test case dependencies are present. Performing this check provided a result of multiple failed runs, which is expected taking into account the stability situation and the amount of test runs needed.

However, the outcome was not always the same when repeating those failed test runs. When the failed runs were tested again by excluding the same tests, all the runs were able to pass. If the exclusion of some single affects towards some other test case, it is hard to prove it since the errors that came up cannot be constantly reproduced. The result of this test method can be taken as an indicator that there are not this kind of dependencies occurring between the test cases where some other test cleans up the state polluted by a previous test. At least not in the original order.

The same script altering was done also for Single Test Group Repeater, changing the required *-sg* command line switch to *-xg* to exclude the test groups of which names contain the given string. This testing is not so time demanding as there are not as many test groups as there are test cases. In addition to that, there are also less test cases in each run, since a whole test group is being excluded. This test would bring up dependencies from similar situation than in single test case exclusion, but at a test group level. The result of this method is similar to the result of single test

exclusion method. Errors were occurring, however any of those could not be constantly reproduced when executing the same run again. The same conclusion can be made: there are not this kind of dependencies occurring between the test groups, at least not in the original order.

7.1.3. Single Test and Test Group Repetition

Utilizing Single Test Repeater with the command line switch *-r* applied with some suitable number of how many times to repeat the tests under the same start up executed one-by-one can be an effective way to discover undesired behaviour from individual test cases. If the test leaves the system in a completely clean state after it has finished, it should be able to run again as many times as the possibly existing limitations from the application allows. If some test leaves something uncleaned, pollutes the state with packets in some buffer for example, there is a possibility of them to be found when repeating the same test under the same start up, if there are checks that are suitable for that or if the uncleaned subjects cause a crash for example.

Single test repetition was performed with multiple different values for how many times to repeat. Some test cases started to fail earlier and some required more repeats for them to start to fail. Some tests survived even when tested with *-r 1000*. The test failures or crashes do not make it clear in every case whether the reason for it is due to a limitation from the application or from an actual resource leak or other misbehaviour. The value of how many times to repeat was set on being 600, since while going above that the number of new crashes or failures did not increase that fast anymore. Single Test Repeater used with the command line switch *-r 600* introduced failures or crashes on 31 different tests from nine different test groups, with 11 different types of errors, test failures or crashes. The types of these failures are presented in Table 5.

Table 5. Failures occurred when repeating single test cases separately, 600 times for test under the same start up.

Error	Type of failure	Test groups	Test cases
Message not received in time	Test failure	1	14
Failed to allocate message	Test failure	2	6
Test failed	Test failure	2	3
Assertion error in Main	Crash	1	2
Undefined message	Crash	1	1
Undefined event	Crash	1	1
Segmentation fault, buffer	Crash	1	1
Segmentation fault, memory	Crash	1	1
Application stuck	Stall	1	1

Similar failures and crashes popped up when performing the repetition test for test groups with Single Test Group Repeater. Since there are more tests on a run because the whole test group is included instead of a single test, the repetition value can be set lower. The number of repetition times for test group repetition method was set to be 60. One test group started to instantly fail on the second time the tests started under the

same start up, providing a clear indicator which was missed from cleaning up in the previous run. Combining the single test exclusion method to this one in the instantly failing test group, the test which was not thoroughly cleaned up was easily located. Ignoring that test case, the repetition for this test group started to pass. All in all, eight out of the 32 test groups crashed or failed to finish the testing due to a failure in some test case, when repeated 60 times under the same start up.

7.2. Utilization of Test Repeater

7.2.1. Test Repeating

Utilizing the Test Repeater -tool was started with the basic usage of the script: repeating the tests multiple times in the original order with a new start up for every run. Using this method executes the tests as they are executed in the real situation in CI or by developers, but many times in a row. This provides an easy way to acquire actual data about the stability situation, which can then be used to analyse whether some change to the code causes changes to the stability situation.

Another important information easily acquired by using this method is what are the test cases that are failing, and whether some change to the code affects that. Without this kind of intensive and centralized testing this information is scattered to different places and most likely among different people as well. Utilizing this method often makes it easier to identify the common factors of those failures and whether there is a clear change in the stability situation after some change a developer is proposing.

7.2.2. Test Order Shuffling

Shuffling the order where the tests get executed is an effective way of bringing up the order dependencies. This can easily be done by utilizing the Test Repeater with the command line switch *-s* which, when given no seed, uses time-based, different seed for each run with a new start up. This method executes tests automatically the given number of times, where every test run has a different order of executing the tests. Compared to test repeating in the original order, this method provides shorter testing time due to the fact that there are more failures coming up which makes the test run to stop earlier. This also introduces different failures and crashes from different situations.

The reproducibility of the failures is also increased as the failures or crashes are most likely occurring due to order dependencies, and so the failure can likely occur again when using the same seed as in the failed run. In the normal testing the failures are occurring randomly and due to that they are hard to reproduce. Using the Test Repeater with new, shuffled test order every time for 200 rounds, 200 log files with test failures or crashes are gathered, since the success percent for shuffled runs is 0%. This provides a lot of important information about the failing test cases and failure situations.

Starting to investigate some failure based on some randomly chosen log from a failed test run can be time-consuming, and the reason for the failure can be hard to locate. Some failures are happening later and some earlier, which also means that the time used

for those test runs vary as well. Most likely the most effective way to investigate, at least based on test run execution time, is to take under investigation the test order which executed the least number of tests before the failure occurred. Choosing a test order where the least number of tests were executed before the failure provides a situation where not so many functions were executed, which helps on the tracking of the reason for the failure. The information on how many tests were executed can be acquired by searching for the number of tests executed from the log files from the failed runs.

When the number of tests executed is acquired, the file with the least number of tests executed can be chosen to be taken under investigation, for example. The seed used to acquire that order of tests can be fetched by searching the log file manually or with a simple shell command. The seeds can also be acquired from the *results.txt* file created by the tool. If the failure is occurring due to an order dependency, it can be reproduced when the tests are run in the same order.

Now that the failure or crash can be reproduced, it is easier to locate the reason why it is happening. The developer can for example use a debugger to get more precise information about the error, and perhaps instantly see what test is polluting the state and which way. However, the debugger does not always provide much clearer information about why the failure occurred compared to the log file. This was the situation in many of the order dependencies took under investigation, so some other procedure needed to be applied to point out the actual cause for the failure.

The principle of binary search was taken into use for trying to pinpoint the test case that causes the failure in the ongoing investigation. In binary search, each iteration cuts the search interval by half compared with the previous iteration [45]. By ignoring approximately half of the test cases that are executed before the failure, the remaining set could be tested to see if the failure still occurs. In practice, taking out precisely the first or the second half of the test cases is not so convenient to do for shuffled test runs, as the tests belong to different test groups. Due to this, most of the time the reducing of test cases was done to whole test groups, keeping the process simpler and easier to manage. This process is illustrated in Fig. 21. In some cases, the cutting out needed to be done at the test case level, by excluding one test at a time. When the test case causing the failure was pinpointed, it could be taken into further investigation.

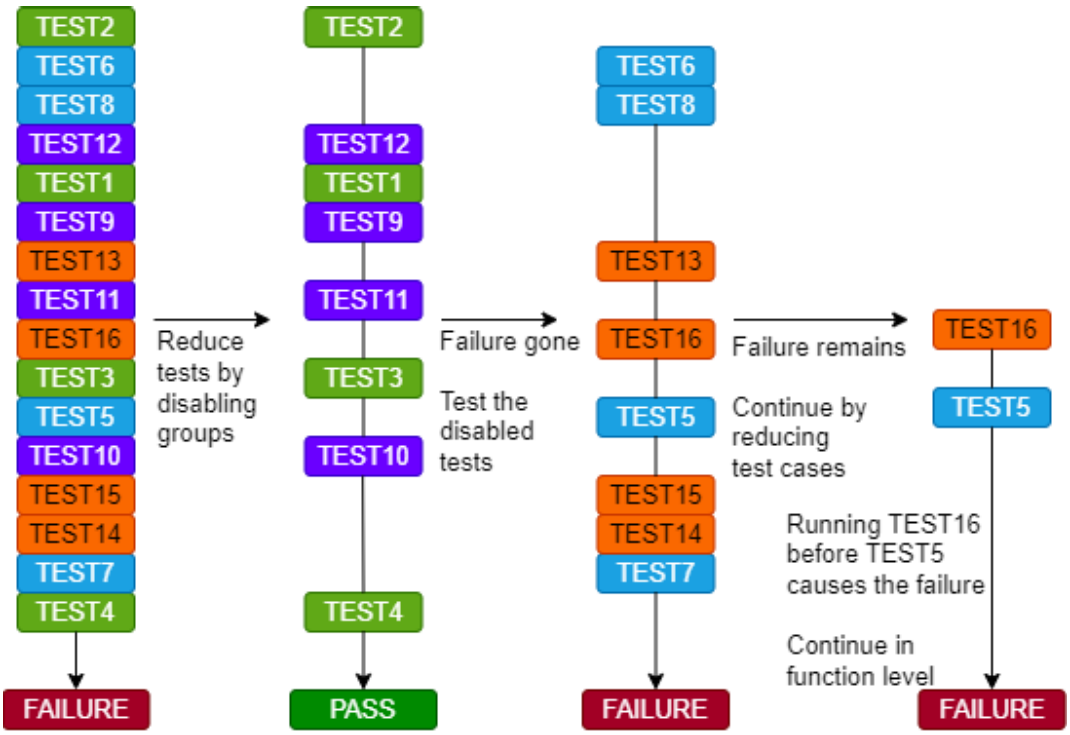

Figure 21. Applying the principle of binary search to a shuffled test run to help to locate the failure. Colors representing of belonging to same test group.

Often, a state-pollution or a resource leak leading to an order dependency is due to lack of function calls in some test group's teardown function, so cutting down the test cases by test group level is more efficient, and it helps to locate the faulty teardown functions as well. The existing order dependencies may have led to a situation where there is lack of function calls in some test group's setup function. There might even be extra function calls in it, which may not be needed and are not cleaned in the teardown function of the test group, so it is important to investigate not only the teardown functions, but also the setup functions.

The binary search type procedure, depending on the structure of the test case, could in some cases be done to the function calls of the test case or to its setup and teardown functions. The applying of binary search method on different levels proved to work quite well. However, the procedure can consume a lot of time, especially after when most of the failures are already fixed. This is due to new failures requiring more tests to be executed because the failures do not occur so often anymore. Most of the order dependencies were found and solved using this method.

7.3. Results

After some problems were found with the tools and methods presented earlier, the stability of the test cases increased noticeably. However, when there are also many developers and teams fixing problems and developing the module forwards all the time, it is difficult to point out which partition of the improved stability is due to the work performed with these methods, and which are from elsewhere. Simultaneously, when

fixing order dependencies, there were some findings from other developers that fixed errors occurring due to timing issues, which had a positive impact on the stability when running the tests in the original order. Test repeating method with the Test Repeater tool was used to validate the findings and that the proposed fix worked.

Before the fix regarding to timing issues, however, the stability of shuffled test runs was also increased from fixes done directly with the tools. Method of shuffling the test order and repeating the tests proved to be useful on reducing the number of order dependencies and bringing up the stability level. In fact, the success percent was brought up to 100% at one point, but due to other big changes to the master repository occurring at the same time, the percent got reduced after pulling the latest version of the code and merging the changes. The results of stability tests after some fixes, performed the same way as the initial stability tests, for the GCC compiler can be seen from Table 6 and for the Clang compiler from Table 7.

Table 6. Statistics from 200 test runs after some fixes for the GCC compiler

Configuration	Tests in a run	Passed runs	Success %
GCC	193	200	100%
GCC All Enabled	224	200	100%
GCC Shuffled	182	153	76.5%

Table 7. Statistics from 200 test runs after some fixes for the Clang compiler

Configuration	Tests in a run	Passed runs	Success %
Clang	196	200	100%
Clang All Enabled	235	200	100%
Clang Shuffled	185	140	70%

7.4. Monitoring Stability and Preventing Dependencies

To monitor the stability of the test cases and to prevent new order dependencies occurring, two automated runs were introduced to the CI pipeline used in this SW development area. Both of the jobs utilize the Test Repeater -tool built for this thesis.

The first job runs the tests 20 times in a row in the original order, with a new start up for each round. Since the stability of the tests running in the original order has successfully been increased to 100%, this job is expected to succeed. However, since the tests are executed 20 times, the job is rather long in comparison with normal jobs that run tests for some module only once. If this job would have been added to the blocking part of the CI, which gets checked for every commit made to the pipeline, it would increase the length and duration of the CI pipeline queue, which would slow down the time to merge. For this reason, the job was added to a part of the CI pipeline which is not blocking, and which gets ran once every night, when the load on the CI pipeline is at the lowest. This means that the outcome of the job must manually be checked every day to ensure that the stability is kept high. Luckily, the CI pipeline

provides a view of the previous runs of the job, so the outcome monitoring and their comparing is very easy to perform.

The other job introduced runs the tests 20 times in a row with a new start up for each run and it runs them in randomly shuffled order using the command line switch *-s*. Due to the stability of a shuffled test run currently being around 70% depending on the compiler, the job is expected to fail. Also, due to the length of the job, this job is not located to the blocking part of the CI either. This job is also located on the part of the CI that gets executed once a night, so this also needs manual monitoring of the outcome. Even though a failure of this job is currently expected, it provides important information about the existing order dependencies between the test cases, as well as information on possible introduction of new order dependencies or state-polluting test cases. Due to the automated run of the job, the developers can pick up the seeds used in failed runs directly from the logs the CI is configured to produce for this job. Also, because the job gets executed nightly, the important information of the current level of stability is continuously available, and the changes that may cause a decrease of the stability can be located rather easily.

8. DISCUSSION

As regression testing is an essential part of the software development process, and it has a big effect on the result of the project, test cases with a nondeterministic outcome are a severe problem in the field of software development. Unstable test cases can have a huge effect on a wide range of subjects, from which the most obvious ones are an increased demand of resources and delays in the project development. In a large organization with a wide range of projects, these things can cross project lines as well. Due to this, flaky tests have gain increasing attention in the field, and more and more tools and methods are being introduced to tackle the issue.

The objectives of this thesis were to build tools that help developers to increase the stability of the test cases by reproducing the errors, locating them, and fixing them to reduce the number of flaky test cases. In addition to that, to gain information about whether the tools are actually useful, the objectives included the usage of the tools for trying to improve the stability of the tests, and to introduce preventative methods by utilizing those tools for helping to keep the stability high. As a requirement, the presented tools and methods should be simple to use and not cause significant delays in daily work.

To meet the objectives, three tools were created, which all of those can be used with different ways to acquire different information. Tools are constructed to help the user to the correct usage of it by printing instructions and example commands. The tools are started from the command line inside the developer's regular working environment, and the arguments use the same build and test commands developers use in their daily work. In addition to that, the user can provide a couple of different arguments to gain a desired testing method and log saving.

The usage of the tools led to different findings in the test codes that improved the stability of the tests when fixed. Comparison of the stability in the initial situation and after some fixes is introduced in Table 8. In a big project like this with a multiple different teams and developers altering the same code base, it is difficult to point out what amount of the increased stability was due to the usage of these tools. However, the rise of the stability level of shuffled test runs was brought up directly from the findings provided by these tools and methods performed with them.

Table 8. Statistics from the initial situation versus the situation after some fixes

Configuration	Tests in a run before	Tests in a run after	Success % before	Success % after
Clang	194	196	82.5%	100%
Clang All Enabled	222	235	58%	100%
Clang Shuffled	185	185	0%	70%
GCC	192	193	100%	100%
GCC All Enabled	220	224	100%	100%
GCC Shuffled	183	182	0%	76.5%

As preventative methods, two jobs were added to the CI pipeline used with this project. One of them runs stability runs for the tests by running them 20 times in a row with a new start up, to acquire information about the current stability daily. Another

one runs the jobs also 20 times in a row with a new start up, but it runs them in a shuffled order. This is introduced to catch the newly, accidentally introduced order-dependencies in the test cases. These jobs get executed once in every night when the load of the CI pipeline is low, to help to transfer the resource usage away from the high spikes occurring at the day time, when the developers are utilizing the pipeline more.

The tools introduced in this thesis demand more resources than some other state-of-the-art tools that utilize different methods than test rerunning. Most of those, however, could not be used with this kind of application under testing since they are designed to work on different platforms. The tools have provided to be useful in fixing order dependencies to improve test stability. However, the whole locating process of flaky test cases is not automatic, so it requires some manual work from the developers utilizing the tools for pinpointing the test case and the fault in it which is causing the failure. Rerunning the test cases in the order the failure existed is rather easy due to the tool offering the used seeds and command to run the tests, which makes the reproducing of the fault much easier, especially if the failure is occurring due to an order dependency.

8.1. Future Work

The tools introduced in this thesis could be developed forward, so that they require less work from the users. The binary search method could perhaps be automated, so the pinpointing of the failed test case could be done automatically by the tool. Another good addition for the tools could be automated rerunning when using shuffling in the test order. This would use the same seed to run the tests again, to see if the failure is flaky or if it can be reproduced by running them in the same order. The tool could for example form two lists, one of which lists the seeds that are causing failures when ran again, and the other one listing the seeds that pass when ran again. The tool could also list the number of tests executed in failed runs before the failure occurred, since focusing on the shortest test runs found in this way might help in the localization of faulty test cases.

When all the order dependencies are fixed, the regular runs performed by the CI pipeline could be changed to run the tests in a shuffled order, or another job doing so could be added. In case of a failed job run in CI, the same shuffle seed could be used on the rerun of the job, to see if the reason for the failure is flaky or persistent. This would be more effective to catch the changes that are accidentally introducing order dependencies between the test cases.

9. CONCLUSION

The objective for this thesis was to introduce tools that help the developers in the process of locating and fixing flaky test cases. Unstable tests increase the resource usage on multiple levels, cause delays even over the project boundaries, and reduce the developers' trust towards the product and its tests. These aspects make the fixing of the problem very important. The thesis introduces the working area these tools are built for, which is completely new, an under development version of eCPRI module to be used as a FH with an upcoming 5G product. After that, some commonly used software development and testing practices are presented, as well as taking a look into flaky tests, their reasons, as well as a glimpse to some state-of-the-art tools designed for trying to tackle the important issue of flaky tests. Motivation and goals for this work are presented together with the initial situation about the development and the stability of the eCPRI module, as well as the development and testing environment used in this project. Following that, the tools built and methods used with them are presented, as well as the results they derived.

Due to the existence of memory monitoring tools in this project, and because multiple factors were indicating the presence of order-dependent test cases, a decision was made to build the tools to concentrate on order dependencies. Three tools were introduced which can be used with different ways to acquire important information about different types of order dependencies occurring between the test cases. The tools also provide an easier way to run tests automatically. The tools can be used either to locate state-polluting test cases, investigate possible errors or to validate the correctness of a proposed change. The main methods the tools are utilizing are test order shuffling and repeating. The tools are easy to use, as they are executed directly from the developers' regular developing environment, they use the same testing commands what developers normally use, and they provide example commands for the developers.

The tools were successfully used to improve the stability of the tests, and to reduce the existing order-dependencies. The work done directly with the tools brought up the success percent of shuffled test runs from 0 % to 70 % or 76.5 %, depending on which of the compilers was used. In addition to the risen stability level, the reduction of order dependencies helps to introduce new test cases. The tools were also taken in use for automatic monitoring of the stability situation, ran once a night, when the resource usage of the CI pipeline is usually low. This has proved to be useful in situations where new test cases are being introduced. For example, some change may also accidentally introduce new order dependencies or decrease the current stability level of the test cases. With the automatic monitoring, these can be caught faster and easier.

These tools, the improvements to the stability level and the reduction of order dependencies reduce the amount of work that needs to be done to solve the remaining issues of flaky test cases. Due to that, the amount of time the developers have to actually develop the module forward is increased. While the tests ran in the CI pipeline do not fail so often, the integration of the proposed changes to the code happens faster and with a lesser usage of resources. The decreased resource usage and increased time to develop the module forwards leads to savings on the costs of the development of this module. The automated runs for the purpose of stability monitoring and catching order dependencies help to locate the changes that have caused a reduction of the stability level or have introduced order dependencies.

10. REFERENCES

[1] Khorikov V. (2020) Unit Testing Principles, Practices, and Patterns. Simon and Schuster.

[2] Mockus A., Nagappan N. & Dinh-Trong T.T. (2009) Test coverage and post-verification defects: A multiple case study. In: 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 291–301.

[3] George B. & Williams L. (2004) A structured experiment of test-driven development. Information and Software Technology 46, pp. 337–342.

[4] Andrea J. (2007) Envisioning the next-generation of functional testing tools. IEEE Software 24, pp. 58–66.

[5] Gyori A., Shi A., Hariri F. & Marinov D. (2015) Reliable testing: Detecting state-polluting tests to prevent test dependency. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Association for Computing Machinery, New York, NY, USA, p. 223–233.

[6] Gambi A., Bell J. & Zeller A. (2018) Practical test dependency detection. In: IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–11.

[7] Qian H.m. & Zheng C. (2009) A embedded software testing process model. In: International Conference on Computational Intelligence and Software Engineering, pp. 1–5.

[8] Selby R. (2005) Enabling reuse-based software development of large-scale systems. IEEE Transactions on Software Engineering 31, pp. 495–510.

[9] Labouardy M. (2021) Pipeline as Code: Continuous Delivery with Jenkins, Kubernetes, and Terraform. Manning Publications Co.

[10] eCPRI Specification v1.0. URL: `http://www.cpri.info/downloads/eCPRI_v_1_0_2017_08_22.pdf`. Accessed in: 24.11.2022.

[11] 3rd Generation Partnership Project (2017) First 5G NR Specs Approved.

[12] Holma H., Toskala A. & Nakamura T. (2020) 5G technology: 3GPP new radio. John Wiley & Sons.

[13] Fletcher B. (2022), Samsung, Qualcomm hit 8.08 Gbps 5G download record. URL: `https://www.fiercewireless.com/tech/samsung-qualcomm-hit-808-gbps-5g-download-record`. Accessed in 31.10.2022.

[14] Nokia (2022), Nokia and Elisa achieve over 2 Gbps 5G uplink speeds on mmWave with Qualcomm solutions. URL: `https://www.nokia.com/about-us/news/releases/2022/06/21/nokia-and-elisa-achieve-over-2-gbps-5g-uplink-speeds-on-mmwave-with-qualcomm-solutions/`. Accessed in 31.10.2022.

[15] Zaidi A., Athley F., Medbo J., Gustavsson U., Durisi G. & Chen X. (2018) NR Physical Layer: Overview. In: A. Zaidi, F. Athley, J. Medbo, U. Gustavsson, G. Durisi & X. Chen (eds.) 5G Physical Layer, Academic Press, pp. 21–34.

[16] Federica R., Alessandro R. & Pizzi S. (2021) 5G NR system design: a concise survey of key features and capabilities. Wireless Networks 27, pp. 5173–5188.

[17] Morais D. (2020) Key 5G Physical Layer Technologies: Enabling Mobile and Fixed Wireless Access. Springer Cham.

[18] Sarikaya E. & Onur E. (2021) Placement of 5G RAN Slices in Multi-tier O-RAN 5G Networks with Flexible Functional Splits. In: 17th International Conference on Network and Service Management (CNSM), pp. 274–282.

[19] Sirotkin A. (2021) 5G Radio Access Network Architecture: The dark side of 5G, John Wiley & Sons. pp. 125–129.

[20] Larsen L.M.P., Checko A. & Christiansen H.L. (2019) A Survey of the Functional Splits Proposed for 5G Mobile Crosshaul Networks. IEEE Communications Surveys & Tutorials 21, pp. 146–172.

[21] Sanfilippo G., Galinina O., Andreev S., Pizzi S. & Araniti G. (2018) A concise review of 5G new radio capabilities for directional access at mmWave frequencies. Internet of Things, Smart Spaces, and Next Generation Networks and Systems , pp. 340–354.

[22] Shinde C.S. (2020) A pragmatic industrial road map for shifting the existing fronthaul from CPRI to 5G compatible eCPRI. In: IEEE 3rd 5G World Forum (5GWF), pp. 297–302.

[23] Liu X. (2022) Chapter 3 - Optical interfaces for 5G radio access network. In: X. Liu (ed.) Optical Communications in the 5G Era, Academic Press, pp. 49–69.

[24] Leung H.K.N. & White L.J. (1989) Insights into regression testing (software testing). Proceedings. Conference on Software Maintenance , pp. 60–69.

[25] Baldwin C.Y. & Clark K.B. (2000) Design Rules: The Power of Modularity. The MIT Press.

[26] Van Schooenderwoert N. & Morsicato R. (2004) Taming the Embedded Tiger – Agile Test Techniques for Embedded Software. In: Agile Development Conference, pp. 120–126.

[27] CppUTest Manual. URL: `https://cpputest.github.io/manual.html`. Accessed in: 31.10.2022.

[28] Zhang S., Jalali D., Wuttke J., Muşlu K., Lam W., Ernst M.D. & Notkin D. (2014) Empirically Revisiting the Test Independence Assumption. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, Association for Computing Machinery, New York, NY, USA, p. 385–396.

[29] Li C., Zhu C., Wang W. & Shi A. (2022) Repairing Order-Dependent Flaky Tests via Test Generation. In: IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 1881–1892.

[30] Micco J. (2017), The State of Continuous Integration Testing @Google. URL: `https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45880.pdf`. Accessed in 31.10.2022.

[31] Gruber M. & Fraser G. (2022) A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It. IEEE Conference on Software Testing, Verification and Validation (ICST) , pp. 82–92.

[32] Luo Q., Hariri F., Eloussi L. & Marinov D. (2014) An Empirical Analysis of Flaky Tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, Association for Computing Machinery, New York, NY, USA, p. 643–653.

[33] Gruber M., Lukasczyk S., Kroiß F. & Fraser G. (2021) An Empirical Study of Flaky Tests in Python. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 148–158.

[34] Valgrind. URL: `https://valgrind.org/`. Accessed in: 31.10.2022.

[35] Eck M., Palomba F., Castelluccio M. & Bacchelli A. (2019) Understanding flaky tests: The developer's perspective. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, p. 830–840.

[36] Alshammari A., Morris C., Hilton M. & Bell J. (2021) FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In: IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1572–1584.

[37] Bell J., Legunsen O., Hilton M., Eloussi L., Yung T. & Marinov D. (2018) DeFlaker: Automatically Detecting Flaky Tests. In: IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 433–444.

[38] Silva D., Teixeira L. & d'Amorim M. (2020) Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 301–311.

[39] Ahmad A., de Oliveira Neto F.G., Shi Z., Sandahl K. & Leifler O. (2021) A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis. In: 28th Asia-Pacific Software Engineering Conference (APSEC), pp. 338–348.

[40] Ziftci C. & Cavalcanti D. (2020) De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 736–745.

[41] Kang B., Kwon Y.J. & Lee R. (2005) A design and test technique for embedded software. In: Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05), pp. 160–165.

[42] CppUTest. URL: https://cpputest.github.io. Accessed in: 31.10.2022.

[43] GoogleTest. URL: https://google.github.io/googletest/. Accessed in: 31.10.2022.

[44] Argparse module, Python. URL: https://docs.python.org/3/library/argparse.html. Accessed in: 31.10.2022.

[45] P.Parmar V. & Kumbharana C. (2015) Comparing linear search and binary search algorithms to search an element from a linear list implemented through static array, dynamic array and linked list. International Journal of Computer Applications 121, pp. 13–17.

11. APPENDICES

Appendix 1 Test Repeater -tool
Appendix 2 Single Test Repeater -tool
Appendix 3 Single Test Group Repeater -tool

Appendix 1 Test Repeater -script

```
#!/usr/bin/env python3

# This script runs all given tests repeatedly
# for given number of times, with a new start up.
# Made to monitor unstability of tests and to
# locate flakiness.

import argparse
import logging
import os
import shutil
import subprocess
import sys
import time

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("Test Repeater")
parser = argparse.ArgumentParser()

starttime = time.time()
RESULTS_DIR = 'out/logs/repeat_results/' + str(int(starttime)) + '/'
numOfFailedRuns = 0
numOfPassedRuns = 0
failedSeeds = ''

def parseArgs():
    parser.add_argument('--single', '-s',
                        help="Cmd to ran once", type=str)
    parser.add_argument('--test', '-t',
                        help="Cmd to repeat, ran X times", type=str)
    parser.add_argument('--repeat', '-r',
                        help="How many times to repeat",
                        type=int, default=1)
    parser.add_argument('--log', '-l',
                        help="Save 'all', 'failed' or 'none' logs",
                        type=str, default='all')
    return parser.parse_args()

def printInstructions():
    print(parser.format_help())
    example = ("""-s "***BUILD COMMAND***" """
               """-t "***TEST COMMAND" -r 10 -l 'failed'""")
    print("Example: tools/test_scripts/test_repeatedly.py " +
          example)
```

```
def saveResults(cmd, dir, log):
    runs = numOfPassedRuns + numOfFailedRuns
    if runs == 0:
        logger.info("\nNo tests executed!")
        return
    resultStr = ("\nPASSED: " + str(numOfPassedRuns) + " TIMES, " +
                 str(round((numOfPassedRuns/runs)*100, 2)) + "%\n"
                 "FAILED: " + str(numOfFailedRuns) + " TIMES, " +
                 str(round((numOfFailedRuns/runs)*100, 2)) + "%\n"
                 "execution time, s: " +
                 str(round((time.time()-starttime), 1)))
    if failedSeeds != '':
        resultStr = (resultStr+'\nShuffle seeds in failing order: '+
                     failedSeeds)
        helpCmd = cmd.split("-s ")
        resultStr = (resultStr +
                    '\n\nTest this manually with command:\n' +
                    helpCmd[0] + '-s ' + failedSeeds.split(' ')[0] +
                    ' ' + helpCmd[1])
    logger.info(resultStr + '\n')
    if log != 'none':
        with open(dir + 'results.txt', 'w') as resultFile:
            resultFile.write(resultStr)

def removeDir(dir, onlyIfEmpty=False):
    if os.path.exists(dir):
        if not onlyIfEmpty:
            shell("rm -rf " + dir)
        else:
            if len(os.listdir(dir)) == 0:
                shell("rm -rf " + dir)

def getSeed():
    cmd = "grep '(?<=seed: )[^ ]*' -oP" + RESULTS_DIR +"templog.txt"
    result = subprocess.run(cmd, shell=True,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.STDOUT,
                            executable=shutil.which('bash'))
    assert result.returncode == 0, "Seed not found!"
    return str(result.stdout).strip("b'\\n")

def shell(cmd, check=True):
    logger.info('shell: %s', cmd)
    result = subprocess.run(cmd, check=check, shell=True,
                            stdout=subprocess.PIPE,
                            stderr=subprocess.STDOUT,
                            executable=shutil.which('bash'))
    if not check and result.returncode != 0:
        logger.info('ignoring failure: %s', result)
    subprocess.run("echo -ne '\n'", check=check,
                   shell=True, executable=shutil.which('bash'))
    return result.returncode
```

```
97  def loopTests(cmd, repeatTimes, dir=RESULTS_DIR, log='all'):
98      global numOfFailedRuns, numOfPassedRuns, failedSeeds
99      tempLog = dir + "templog.txt "
100
101     shell("mkdir -p " + dir)
102     for i in range(repeatTimes):
103         statusMsg= "Round " + str(i+1) + "/" + str(repeatTimes)
104         logger.info(statusMsg)
105         result = shell(cmd + " > " + tempLog, check=False)
106         if result == 0:
107             numOfPassedRuns += 1
108             if log == 'all':
109                 shell("mkdir -p " + dir + "passed/")
110                 saveLog = "mv " + tempLog + dir + "passed/log"
111                 shell(saveLog + str(i) + ".txt")
112             else:
113                 shell("rm " + tempLog)
114         else:
115             numOfFailedRuns += 1
116             if '-s ' in cmd:
117                 seed = getSeed()
118                 failedSeeds = failedSeeds + seed + ' '
119             if log != 'none':
120                 shell("mkdir -p " + dir + "failed/")
121                 saveLog = "mv " + tempLog + dir + "failed/log"
122                 shell(saveLog + str(i) + ".txt")
123
124 def main():
125     args = parseArgs()
126     shell("mkdir -p " + RESULTS_DIR)
127     if args.single != None:
128         shell(args.single + " > " + RESULTS_DIR + "buildlog.txt")
129     try:
130         cmd = args.test
131         if "-s'" in cmd:
132             cmd = cmd.replace("-s'", "-s '")
133         loopTests(cmd, args.repeat, log=args.log)
134         saveResults(cmd, RESULTS_DIR, args.log)
135         if args.log == 'none':
136             removeDir(RESULTS_DIR)
137     except TypeError:
138         printInstructions()
139     return numOfFailedRuns
140
141 if __name__ == '__main__':
142     sys.exit(main())
```

Appendix 2 Single Test Repeater -tool

```
#!/usr/bin/env python3

# This script runs all tests separately
# for given number of times.
# Made to locate unstable tests.

import logging
import sys
import time
from test_repeatedly import loopTests, parseArgs, removeDir, \
                            shell, saveResults, parser

logging.basicConfig(level=logging.INFO)
log = logging.getLogger("Single Test Repeater")

starttime = time.time()
RESULTS_DIR = 'out/logs/single_test_results/' +
              str(int(starttime)) + '/'
TESTS_FILE = RESULTS_DIR + "tests.txt"

def getTestnamesToFile(cmd):
    pathToFile = RESULTS_DIR + "testList.txt"
    shell(cmd.replace('-sn', '-ln') + " > " + pathToFile)
    shell("grep '^[^0-9]' " + pathToFile + " | grep ' Test' | " +
          "sed 's/[[:digit:]]\+:.*//' | tr ' ' '\\n' > " +
          TESTS_FILE)

def printInstructions():
    print(parser.format_help())
    example = ("""-s "***BUILD COMMAND***" """
               """-t "***TEST COMMAND*** CPPUTEST_ARGS='-sn'" """
               """ -r 2 -l 'failed'""")
    print("""\n\nNOTE: Test command must have '-sn' argument:
          CPPUTEST_ARGS='-sn' \n""")
    print("Example: tools/test_scripts/single_test_repeater.py " +
          example)
```

```
34 def main():
35     args = parseArgs()
36     shell("mkdir -p " + RESULTS_DIR)
37     if args.single != None:
38         shell(args.single + " > " + RESULTS_DIR + "buildlog.txt")
39     try:
40         if '-sn' not in args.test:
41             raise TypeError
42         getTestnamesToFile(args.test)
43         cmd = args.test.split('-sn')
44         with open(TESTS_FILE, 'r') as tests:
45             for test in tests:
46                 test = test.replace('\n', '').split('.')
47                 testCmd = cmd[0] + ' -sn ' + test[1] + cmd[1]
48                 testDir = RESULTS_DIR + test[1] + '/'
49                 loopTests(testCmd, args.repeat, testDir, args.log)
50                 removeDir(testDir, onlyIfEmpty=True)
51         saveResults(args.test, RESULTS_DIR, args.log)
52         if args.log == 'none':
53             removeDir(RESULTS_DIR)
54     except TypeError:
55         printInstructions()
56 
57 if __name__ == '__main__':
58     sys.exit(main())
```

Appendix 3 Single Test Group Repeater -tool

```
#!/usr/bin/env python3

# This script runs all test groups separately
# for given number of times.
# Made to locate unstable test groups.

import logging
import sys
import time
from test_repeatedly import loopTests, parseArgs, removeDir, \
                            shell, saveResults, parser

logging.basicConfig(level=logging.INFO)
log = logging.getLogger("Single Group Repeater")

starttime = time.time()
RESULTS_DIR = 'out/logs/single_group_results/' +
              str(int(starttime)) + '/'
TEST_GROUPS_FILE = RESULTS_DIR + "testGroups.txt"

def getGroupsToFile(cmd):
    pathToFile = RESULTS_DIR + "testGroupList.txt"
    shell(cmd.replace('-sg', '-lg') + " > " + pathToFile)
    shell("grep '^[^0-9]' " + pathToFile + " | grep ' Test' | " +
          "sed 's/[[:digit:]]\+:.*//' | tr ' ' '\\n' >" +
          TEST_GROUPS_FILE)

def printInstructions():
    print(parser.format_help())
    example = ("""-s "***BUILD COMMAND***" """
               """-t "***TEST COMMAND*** CPPUTEST_ARGS='-sg'" """
               """ -r 2 -l 'failed'""")
    print("""\n\nNOTE: Test command must have '-sg' argument:
          CPPUTEST_ARGS='-sg' \n""")
    print("Example: tools/test_scripts/single_testgroup_repeater.py
    " + example)
```

```
def main():
    args = parseArgs()
    shell("mkdir -p " + RESULTS_DIR)
    if args.single != None:
        shell(args.single + " > " + RESULTS_DIR + "buildlog.txt")
    try:
        if '-sg' not in args.test:
            raise TypeError
        getGroupsToFile(args.test)
        cmd = args.test.split('-sg')
        with open(TEST_GROUPS_FILE, 'r') as groups:
            for group in groups:
                group = group.replace('\n', '')
                testCmd = cmd[0] + ' -sg ' + group + cmd[1]
                testGroupDir = RESULTS_DIR + group + '/'
                loopTests(testCmd, args.repeat,
                          testGroupDir, args.log)
                removeDir(testGroupDir, onlyIfEmpty=True)
        saveResults(args.test, RESULTS_DIR, args.log)
        if args.log == 'none':
            removeDir(RESULTS_DIR)
    except TypeError:
        printInstructions()

if __name__ == '__main__':
    sys.exit(main())
```