OAMK
OULU UNIVERSITY OF
APPLIED SCIENCES

Kalle Soukka

AUTOMATED TESTS IN AUTOMOTIVE SOLUTIONS

AUTOMATED TESTS IN AUTOMOTIVE SOLUTIONS

Kalle Soukka
Bachelor’s Thesis
Spring 2023
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author(s): Kalle Soukka
Title of the bachelor's thesis: Automated tests in automotive solutions
Supervisor(s): Olli Himanka
Term and year of completion: Spring 2023 Number of pages: 28

This thesis is a look into a customer project in doing automated tests in automotive software. It details a time period of three months during which the project took place. This project was done mostly remotely by using remote computer methodologies to control the testing devices.

Testing was done using Profilence company Tau analysis tool and the system tested was an Android based. The results of tests were saved in cloud and then analysed so that the customer would get a picture of the problems appearing in the software.

Throughout the project there were also many different changes to both the software and the hardware and new tests, and test methodologies had to be created for these. There was also a lot of maintenance that had to be done for the devices throughout the project. The use of devices continuously encountered many different problems during the test runs and new software and hardware parts needed new implementations of testing.

Analysing shows that the stability of the software was improved in many different fields of the software. While all the customer requirements were not achieved, the software was improved

Keywords: Test automation, Automotive, Android, Apple CarPlay

PREFACE

This thesis was completed while working in company Profilence using their analysis tool Tau to execute the tests. The goal was to analyse a customer project of 3 months of doing automated tests in automotive solutions. Supervisors for this thesis were Sami Utriainen from Profilence, working as CTO and assisting person from the company, and Olli Himanka from OUAS.

Oulu, 28.3.2023
Kalle Soukka

CONTENTS

ABSTRACT 3

PREFACE 5

CONTENTS 6

VOCABULARY 7

1 INTRODUCTION 7

2 SETTING UP TEST ENVIRONMENT 9

2.1 Functional testing vs non-functional testing 9

2.1.1 Test case structure 11

2.2 Memory leak problem and mobile device problems 12

2.3 Setting up testing stations at client’s site 15

2.4 Working with the test stations remotely 17

3 UPDATING SCRIPTS AND TESTING APPLE CARPLAY 18

3.1 Phone problems 18

3.2 New features with Apple CarPlay 20

4 FIRST TEST RESULTS 22

4.1 The memory leak problem 23

4.2 Bluetooth and location mocker problems 23

5 ISSUES WITH CAN BUS DEBUGGER AND SOFTWARE SLOWDOWNS 25

5.1 Issues observed later into the project 26

6 CONCLUSION 28

REFERENCES 30

VOCABULARY

ADB = Android Debug Bridge

HMI = Human-Machine Interface

Tau = Profilence test tool

GPX = GPS exchange format

HVAC = heating, ventilation and air conditioning

USB = Universal Serial Bus

UI = User Interface

ANR = Application not responding

ADB shell = software used to send commands to android device

Dumpsys = ADB shell command outputting data of different system parts

1 INTRODUCTION

In the last few decades, the world has started to see the rise of smart devices. Smartphones were the first device category to see this change and since then more different devices have become smart. From household appliances such as fridges to health inspection devices we start to see the rise of smart devices everywhere. This has also been the case with automotive software. Before, automotive software has been quite simple. However, in the recent years, they have started to support full operating systems with a lot of different functionalities and features which could earlier be found only in smartphones .

These categories have gone through phases of development. In the beginning all of them have been quite unstable and problematic due to lack of experience and knowhow that comes with experience of working with the technology.

Automotive industry has gotten smart device integration only recently and is still in its starting phase in terms of development. Some people say that automotive software is very fragile and complex to develop [1].

Testing a device or software is key part of device development. When it comes to testing automotive software, the methods are still quite exhausting. Often, the tests are executed while driving the car. This approach to software testing is workable but has many severe flaws. Field tests are expensive to produce both financially but also in manhours and might also be risky to perform depending on the conditions. This way of testing is quite excessive as well considering that there is a lot of extra activity compared to what is being tested.

With handheld devices and other smaller device categories it is common to have tests executed in laboratory environments This approach fixes the problems mentioned above while also adding the element of test performance; tests can be executed much faster and reliably when the results can be analysed almost immediately after the execution . This is beneficial not only to the client to see more quickly what is wrong with the software but also for the test creator who can see if these tests provide good enough data and if they even work to begin with.

Profilence analysis tool Tau allows testing on laboratory environment with other tools to help analyse the data that is being collected. It can be used with any Android device since it uses the Android Debug Bridge (ADB) for communicating with the device. The tool searches for UI elements in Android and then in the code those elements can be called and interacted with.

Tau tool analyses the device resource usage and plots graphs from those statistics while also observing any irregular activity in the software. It also records different kinds of logs of what is happening on the device which can be used to see what exactly is happening in the device at any given moment.

Customer of this project is an anonymous company doing car Human-Machine Interface (HMI) software for automotive. Their product had already been shipped to the users; however, it had received poor reception. While the

customer had their own tests most of these were functional tests which did not get to the root of the problems that the users had reported.

The project is a collaboration between three companies where Profilence would provide the test tools and a partner company would have one of their employees on site at the client's office to provide support. The devices can be configured remotely. However sometimes the devices had to be fixed on-site if for example the physical connection or ADB caused problems. And since the test setups had to be done locally at the client's site, somebody had to be there on-site to help.

The requirement of the customer was to have the Profilence testing and analysis tools to find out problems of their software. On top of this the requirement was to have at least 90 test cases. Before the project there were about 30 cases already done for the automotive. The customer also wanted to have direct discussions between them and a test engineer.

Three people were assigned to this project: the test engineer Kalle Soukka, Profilence CTO Sami Utriainen and a person from previously mentioned partner.

2 SETTING UP TEST ENVIRONMENT

The first task of this project was to set up the test stations. The client's test rigs were not mobile, so these test stations had to be set up locally at the client's site. This meant a week's trip to the client's site to get everything ready for testing that would be done there.

2.1 Functional testing vs non-functional testing

The types of testing that can be done to software can be categorised into functional and non-functional testing. Functional testing is the activity of checking that the software works in the first place and that the behaviour of the software

is correct. Non-functional testing is testing the performance of the applications so that it matches the requirements [2].

Usually, functional testing is easy to produce and can usually be done without additional tools. Non-functional tests are harder to create since it requires a way to analyse background elements such as computer parts. Due to this non-functional testing usually requires a separate platform to run them.

For automotive software, testing is usually functional. Non-functional tests are usually performed in laboratory conditions and automotive software is usually tested in field tests. This does not mean that non-functional tests are not done for automotive software however they are more uncommon.

Tests can also be executed in what is called black box, grey box, or white box testing environments. Black box testing means that the tests are executed on a platform where code of the software, implementation details and other aspects are not visible when creating tests. These elements are things such as the internal structure of the code. White box testing is testing where things such as those mentioned are known to the tester. Gray box testing is the middle ground between these where some parts are known while others are not [3].

Tau tool can do functional testing as well; however, it is used more commonly for non-functional testing and analysis. It can also be used for any of the information levels. However, in this project testing was done in black box environment. Tests are written in Python programming language using Python language programming libraries which are used with Tau tool along with some C programming language-based libraries.

There are a few different levels of testing depth. Testing can be started from creating tests which test if the software works in the first place, and the testing depth can go to types of tests that are executed in this project. In addition, there are other types of software testing that were not tested even though they could have been implemented. These include testing such as smoke testing or monkey testing [4].

The type of testing done in this project is stability testing. Stability testing is part of reliability testing. The difference between them is that in reliability testing on top of testing the stability it also tests how long does it take to recover to normal state if problem had been encountered. The goal of stability testing is to gather analysis data of different components and see their behaviour. This reveals if certain applications or other processes are using certain resource to the extent that it would cause problems.

2.1.1 Test case structure

The tests are executed by creating test cases which would resemble the real-world use cases of the software. A typical driver drives their car around 50 kilometres which is usually the distance to work and back [5]. The driver might start playing some music or adjust the heating, ventilation, and air conditioning (HVAC) or might download a new application. These actions are reflected in the test case where before the actual test Tau does some of these actions at random. All of these must be made while the car is stationary, so this is also reflected in the test case. This structure is visualised in FIGURE 1.

The main feature tested in the test cases is navigation; this would be the main feature used in the HMI. Most test cases created would centre around the maps application where they would first do something else before starting the actual navigation. These other things include things such as putting on some music or rearranging some icons in the UI. There are some test cases where a single application is tested without navigation; however, the car is still set to drive.

For navigating, Tau installs a process to the device that mocks the location of the device. Normally the device would use the current location of the device. However, with this process this location can be changed to be anywhere in the world and thus the device can be put to drive anywhere. For the routes used in the test cases, there were a few premade routes that the test randomly selects from. Tau also allows the creation of maps GPS exchange format (GPX) files which can be used to create new curated routes. The desired speed of how fast these routes would be driven in can also be configured.

Some pre-test case set up can also involve simulating the device going to sleep mode. This can be done through software where the device is put to sleep but not rebooted, then wait some time and then wake the device. There is also a device part that can simulate this but this is not widely used in this project because the device was set up very late into the project as well as the fact that using it would cause the screen to permanently go dimmer. Despite these problems this part did get some testing and most of the sleeping was done through software.

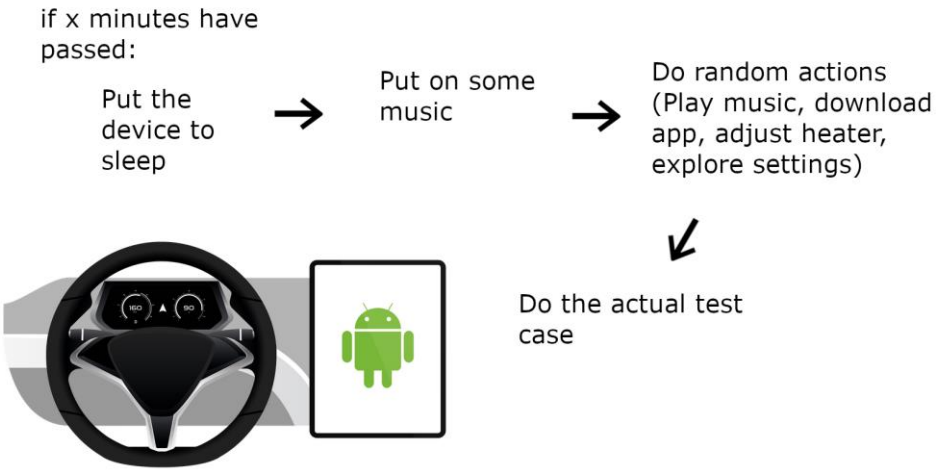

FIGURE 1. Visualisation of the test case structure

2.2 Memory leak problem and mobile device problems

During the project several difficulties were faced. The first one was a memory leak problem . Memory leak is a problem in software where a certain application or process takes memory system resource so much that it is problematic. Usually, the system cleans up memory so that it would not start causing problems. However, sometimes the memory usage can be higher than usually in one process. Sometimes the process can just continue using higher and higher amounts of memory without ever cleaning up previously used memory thus starting to cause problems. Often the main indicator of memory leak is the fact that the system keeps rebooting every now and then due to not having enough memory for critical system processing. Another good indicator is UI slowing

down; since the system does not have enough memory it usually slows down other applications since they do not have enough memory for their own functionality. These problems typically go away after reboot only until some process starts memory leaking again. The way the memory increases can be seen from FIGURE 2.

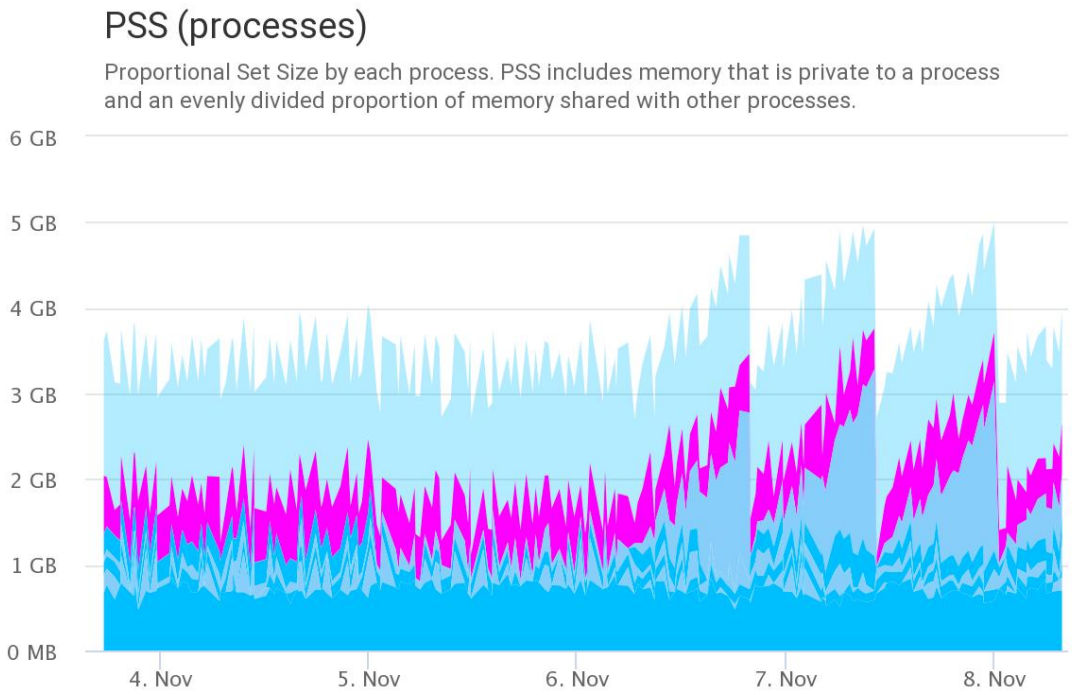

FIGURE 2. Visualization of the memory leak problem. Memory usage per time

The second problem noticed was related to phone calling via ADB shell. ADB shell is a way to send commands to an Android device. After the phone would initiate a call to another phone, a value in the phone, called mCallState, changes. The state zero is when there is no call active; state one is when the device is being called to and state two is when the call is active. The problem was that when running the command to get the mCallState value it would give out two of them. One of these values would stay at mCallState 0 while the other one would change to the correct value.

Later in the project tests for Apple CarPlay had started and its configuration had some difficulties. How Apple CarPlay works on the device is that it is an Android application that works when an iPhone is connected to the device via USB. This alone created its own challenge since normally the device is connected to the

PC via USB which would not work here due to the iPhone requiring the USB connection. ADB connection can also be wirelessly over Wi-Fi or also over ethernet, even though doing it over ethernet is uncommon since most android devices do not have an ethernet connector. For this specific case ADB over ethernet was chosen since it was easier to set up. The client also had instructions to get this working on the testing device; however, this turned out to still have some problems. Firstly, to get everything working the device must be set so that it is visible over the internet via setting a property on the device. After this process, the address of the device on the internet is given to ADB. After this process the one parameter that the device has needs to be changed. In android devices there are two text editors that can do this job, sed and vim. Either one can be used but what also needs to be done is to set correct properties to this specific device file. This also needs to be done for the right connection since now on ADB both, the local USB connection and the wireless connection are visible on ADB. This turned out to be something that took a long time to figure out; connecting to the wrong connection did not allow to change the read and write properties of the file. After all this setup both the iPhone can be connected via USB and the test device can be accessed on ADB, which the visualisation to can be seen from FIGURE 4.

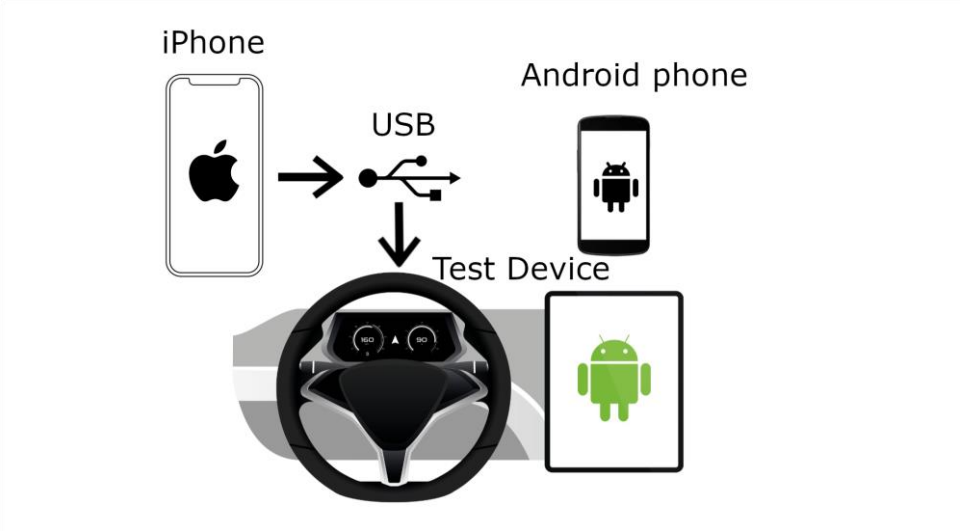

FIGURE 4. Visualisation of the connection when using Apple CarPlay

2.3 Setting up testing stations at client's site

Before the trip some preparations had to be made. Since testing machines had to be carried to the customer's office, they had to be prepared so that there was as small amount of on-site configuration as possible. This meant downloading necessary applications and tools and configuring the remote desktop connections to the device.

Out of some options among remote desktop solutions Chrome Remote Desktop was chosen. This decision was made because most other options would require holes in firewall and other preparations to make it work. Chrome Remote Desktop works by installing software to your computer and then all the devices with the same chrome account can now access this desktop remotely. This allows a relatively easy and reliable connection setup.

Other preparations included preparing two phones to use with the actual testing device. One of the phones would be connected to the testing device via Bluetooth to be used as the main phone call device and to play audio through Bluetooth to the device. The other would serve as a phone receiving phone calls and speak voice assistant commands.

While the actual device to be tested was a closed environment, there was an emulator of the device that some of the preparation work could be done. This emulator was used to get familiar with the system. Some scripts were able to be created this way as well.

The start of the project included a weeklong trip to the client's site. In order to prepare the test setups and to have conversations with the team at the client. On the arrival the test devices were prepared so that the setups were able to be built immediately. However, when preparing the setups some problems arose. One problem was that the USB connection to the device had to be done with high-speed USB. Some of this was able to be averted with USB-hubs but this still did not fix the problems entirely. These USB problems would become a trouble throughout the project.

The customer had two different types of software to test with different specifications. The test scripts worked only with one of the software versions, so scripts had to be made for the other device. On top of this some scripts had to be modified with the other software version as well. The customer also wanted to implement their own test cases into Tau tool.

These test stations would also be periodically updated with new versions. The version update cycle was not consistent so sometimes updates would come a couple times a week. The customer also wanted to test other elements of the system that would require some additional parts or elements. These would be connected to the computer via serial port connections and would be controlled that way. However, some parts were not able to be controlled through ADB or serial ports at all so other methods had to be created.

The week went by with fixing problems in the scripts while also talking with teams on how their section could be tested. During this time there were not any additional devices left for creating and managing test scripts so those had to be modified in between test runs. For the second software there were not any additional test stations. This meant that maintenance was difficult to do and had to be done during downtime.

Since Tau tool reads Android UI elements most of the test cases use commands to utilise these elements. However, some parts of the software did not have any elements to find, and the second software did not have any elements to find at all. This created some challenges where unconventional ways to control the device had to be done. There were also some problems with the second software with not having Google services set up correctly to the device so some test cases which used applications downloaded from Play Store had to be either removed or those apps had to be downloaded some other way. Spotify was downloaded to the device by having the apk package of it and downloading that to the device directly from its ADB shell.

Other problems that arose were faulty system parts mainly regarding the internet connection of the device. Controlling the additional parts of the device

through serial port also provided some challenges. These problems were mostly fixed by the end of the week.

2.4 Working with the test stations remotely

After spending the week in the client's site, the development for the project continued in a different way. A person from the partner company continued to stay at the client's site so that if physical problems were to rise, they would be the one to fix that. The main development would be done in Profilence office at Oulu by remote connecting to the devices running in the client's site. Since there still was not a dedicated scripting computer, fixing test code still had to be done between test runs. New test cases had to be developed as well. In the requirements of the customer the amount of test cases required was 90 while there were only around 30 test cases done so far.

Building new test cases proved to be difficult. This was due to the semi limited nature of the tested software. In other product categories the number of testable elements in the software is often larger due to the app store giving many different possible test cases. While Android Automotive has the Play Store it does not contain many applications in it to test. Many of these also require things such as the registration plate of a car to access which in the test environment there is not. The core of the software is also quite limited in its capability. The web browser cannot be accessed much and while there is the navigation application and radio to test these do not provide too many test cases either. Regardless, some applications were able to be made into test cases and some new UI test cases were also created throughout the project.

Some low-level development was also done. As new parts were added to the device and wanted to be tested, a way to control those parts had to be developed. This happened through serial ports of the device from where commands would be sent to the device parts. This also meant that discussions with the client had to be held since these commands were from there.

During the first few weeks another type of activity was to gather results of the test runs. There were many interesting findings, some of which were visible almost from the very beginning.

3 UPDATING SCRIPTS AND TESTING APPLE CARPLAY

During the first few executed tests the memory leak problem was found. This issue took a few weeks to sort out. After this issue was solved, some other problems started to surface even more than before. Along with the memory leak problem being solved, new elements arrived at the project. There were some new system parts to test; some completely new software elements began to be tested and some new problems started to pop up in existing parts.

3.1 Phone problems

The android phones that were chosen for the project ended up having some problems of their own when it came to phone calling. Firstly, the script that was used to make the phone calls uses two different types of versions of it depending on what the case is. If the phone call is initiated from the testing device, then this action would be done through the UI. If one of the reference device phones initiates the call and calls to the testing device this is done through ADB shell of the device. However, in both cases the problem that arose was about verifying the phone call. This meant that method of checking the state of the phone while calling had to be changed. This was solved by returning a string of the output of dumpsys ADB shell call which then would be converted into a list of all the

numerical values that this string has. This list then would be checked if it contains the wanted values of mCallState.

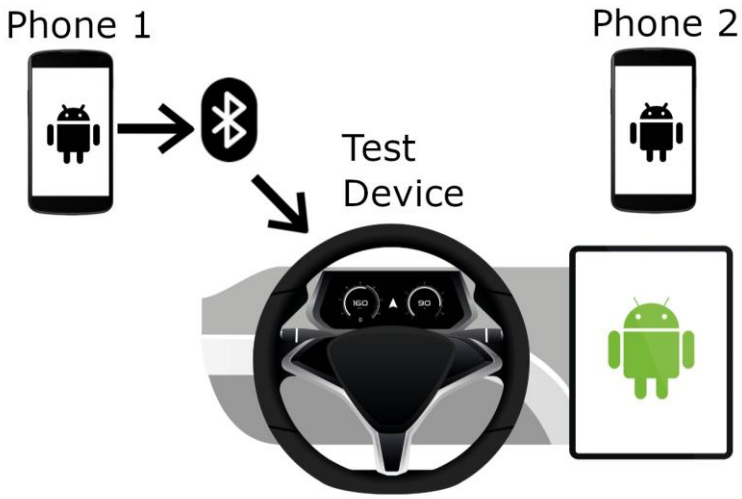

FIGURE 3. Visualisation of the test environment

The mCallState problem was not the only problem related to phone calling. The mobile plans that the phones had seemed to cause problems with phone calling as well. When the phone call would be initiated sometimes it would not go through at all. The call would happen for a few seconds after which the call would be killed on the caller end. For this issue there was nothing that could have been done through the code since this was an issue with either the phone or the mobile plan that was in these phones. However, these phones had been tested before and these phones did not show similar problems during this time, so this issue seemed more likely to be a problem with the mobile plan or the combination of this specific phone and mobile plan. This did not seem a Bluetooth problem since this issue happened even when the phone was not connected to the testing device.

The problem was still fixable though not by conventional or efficient means. The phone would return to calling if it was rebooted which can be done either manually or calling reboot from ADB shell of the phone. This is not an ideal situation since if, for example, the phone calling would start failing in the middle of the

night many of the phone calling test cases would continue failing over the night. Additionally, while the devices can be rebooted through ADB shell and ADB shell commands can be called in the test cases, doing so might cause some problems in the test cases. One of the more common ones would be that the device order would change upon reboot since Tau tool would recognise the other phone first and then would make that the incorrect device for that situation. The only way to fix this would be to either completely stop the test run and switch the device order that way. The other option is to reboot again and hope that the devices connect in correct order this time. Neither of these options is a desirable situation.

In the end, the phone calling was in the test set however the phones were usually left to be without fixing this issue since taking care of this would be a lot of maintaining and usually would cause problems as well and the test runs did not want to be stopped just for this.

3.2 New features with Apple CarPlay

Throughout the project one of the goals was to get the test case amount to 90 cases. This meant that a lot of new cases had to be made. The problem with automotive software is that it is quite limited in what it can do. For example, in phones, things such as web browser and app store open a lot of possibilities in terms of test cases. While these do exist in automotive software their functionality is much more limited. In the web browser, only few web pages can be entered, and many of the links open a pop up to a link of the web page that can then be accessed on some other device. For app store, the situation is a little better however there is not too many applications. On top of this many of the applications might require things such as the car license plate or other aspects which are not present in the test environment. So, in the end only a couple of selected applications could be tested after all. Some already existing applications also got some updates but there was not too much to be done with these. This was due to many of these applications being quite limited in their capabilities as well. Regardless of these limitations new test cases were able to be created for different applications though with limitations.

Another new part to the testing was new sections that the customer wanted to implement and test. The customer also had some of their own tests, many of which were able to be incorporated into Tau tests. One of the biggest new parts to test was Apple CarPlay. After the setup had been figured out the second problem of testing CarPlay arose. Due to the way how CarPlay works, which is Android application that gets data from iPhone, controlling this was quite a problem. The way Tau gets all the screen elements does not work with CarPlay so everything about the test cases planned had to be rethought. There were two different ways how CarPlay would be controlled. One of these would be to just use location coordinates all over the place. However, the limitation of this is that if the UI becomes slow and there is no way of verifying this and thus the coordinates tapped would not go through correctly. Another way that was figured out was to take a screenshot of the current situation, then have a screen reader read the screen content and then use this information in Tau. The downside of this approach is that reading the screen is quite slow, usually taking ten seconds or more to read all the contents. On top of this, the reader can be unreliable as well sometimes reading two separate contents as one or taking different UI elements and thinking that these are letters. In the end, combination of these both approaches ended up being used. For some parts, such as verifying correct elements were on screen and some more critical UI elements, the screen reader method was chosen. For others where screen reader did not work as intended, such as typing on the keyboard or dialler, the hardcoded locations were used.

Additionally, what made CarPlay hard to test was the fact that it works through iPhone that cannot be controlled through methods used here. This created a few problems. Firstly, this meant that using phone calls had to be done through harder ways than if two android phones would be connected. Verifying call state had to be done through the screen reader which, as established earlier, is unreliable. Additionally, navigating through the UI of CarPlay to initiate call was difficult since calling the correct number was unreliable. The navigation did not work either. Normally, on Android side navigation would use location mocker to simulate moving the device through roads. CarPlay would get the location from the location data of the connected iPhone. There were some attempts to fake the

location data on the iPhone however nothing ended up working and this idea was totally scrapped along with testing the CarPlay maps. The main functionality of the CarPlay is the navigation so this was a large setback on testing.

The tests that were made for CarPlay ended up mostly being some phone calling cases and using Spotify media application to play some music through CarPlay. With Spotify, the login process had to be done manually through iPhone even though this process had to be done only once when this setup was done for the first time. After this the rest of the Spotify application was able to be used through the CarPlay UI. Some integration of the CarPlay UI and normal Android Automotive UI was also made. Main test case in this was to first set some music to play through Android Automotive UI after which music would start to be played through the CarPlay UI.

Some script updating was also completed during this time. Main place where the script updating was done was in methods relating to navigating around the system UI through Tau tool. In Android, there are commands called KeyEvents which are actions when a certain button is pressed. There are many KeyEvents many of which are not usually found in everyday phones. Some of these events could be changed to actual hardware calls instead of Android software events. Some things related to location mocker could also be replaced with these hardware event calls.

4 FIRST TEST RESULTS

Even in the first few days of running tests, some of the key problems were found in the system. One of the key problems that arose very quickly was a severe memory leak in one of the software components. This caused the system to reboot every so often. This memory leak also prevented many other problems from showing up in the results since the system did not have enough time before it would reboot to show these problems.

4.1 The memory leak problem

When trying to solve the memory leak problem the first step was to find out what caused it. This was found immediately since the data collected is sent to a server where it can be found which process in the system causes issues. After finding the memory leaking process it was time to get the thing sorted out with the client. Finding the solution to the problem took a few weeks during which time the other issues were not too visible. The system kept rebooting which caused an additional issue where many of the other problems of the system would not show up because the system was not able to be on for long enough for these problems to show up.

These memory leak issues also caused additional problems that otherwise would not be seen. Main issue that was seen was different applications having Android Application Not Responding (ANR) errors all around. ANR is triggered when an UI thread is blocked on android process for too long. So, in other words if the UI does not respond to an action from the user fast enough ANR is triggered. Another problem that is seen is Android frozen frames which is almost the same as ANR; however, how it differs is that frozen frame triggers after 700 milliseconds while ANR is triggered after a few seconds of application not responding. Occasionally, some processes would also end up crashing due to the memory leak problem on the other process.

The main application affected by the memory leak was Google Maps. Maps by itself is already heavy on memory and another process memory leaking makes the situation even worse. Due to this many errors of type java.lang.OutOfMemoryError were seen on maps. This means that in Java language a functionality called garbage collector cannot free enough memory for some application or process to run fully [6].

4.2 Bluetooth and location mocker problems

While the memory leak made other problems less visible there were still some that were noticed early on. One of the main things seen already around this time was crash in Bluetooth process. Unlike memory leak problem this one had

more clear reason for it. The Bluetooth crashed every time the device entered sleep mode, so this was not related to the memory leak. This created a few problems. Firstly, this meant that every time Bluetooth crashed the device lost connection to the phone that was connected to it via Bluetooth. Secondly, this caused all the test cases regarding phone calling to fail thus not making some test cases run to the end correctly. This would end up being a problem all throughout the project

There were also some test tool related crashes on the device. Since Tau installs some processes to the device some of these would also cause some crashes and problems. One of these is the process used to mock the location of the device. Another problem that was seen already around this time, even though went away quickly, was the crash of the main Tau process in the device. For this problem the explanation was not clear at all. It seemed like a problem with the system memory being low however this explanation did not add up. Another reason behind this could have been the USB connection speed being too slow; however, later in the project the device was connected via ADB over ethernet, yet the problem persisted.

Tau gives out a reliability score for the testable device. This is given on the basis of how many problems there were found and what the problem was. For example, a system reset reduces the reliability score by ten while an application crash reduces the reliability score by two. During the memory leak problem, the reliability scores of the runs were low. Some reliability scores were higher than others though that was mostly when the run was short. When the run was going on for a few days ~~then~~ the reliability score would get lower.

5 ISSUES WITH CAN BUS DEBUGGER AND SOFTWARE SLOW-DOWNS

As the project started to come close to the end date the testing continued in the same way as before. Around this time the scripts did not see too many changes, only smaller fixes to some problems observed and some edge cases. The main section of focus around this time was to observe some of the more recent crashes and problems in the test device.

Around this time the testing for the second type of software had stopped. This gave time to focus more on the more critical software to test. This software was experimental and not too urgent. Only three different versions of this software were tested. Some problems with this software were crashes in radio related things. This software did not have Google services due to lacking verification so because of this many of the tests were not too in depth. Additionally, this software kept having a popup about this which then would make the test case fail since the test case could not find correct elements. This popup did not have any identifiers either, so this notification proved to be quite unreliable. In the end, the customer decided to stop testing this software.

For the main software being tested, many of the previous problems continued to persist. Phone cases would keep failing due to the phones not making phone calls and Bluetooth crashes kept happening on the device. The customer had two different versions of this software and while at the start of the project both software versions were tested, in the end one of these versions stopped to be tested due to repeatedly getting the same issues to pop up and cause problems.

Even though there were two rigs of the same version they had different tests running on them. Usually, on one of the rigs a normal test run was done of the current version being tested while the other had some special software or hardware tested, like CarPlay or another way of doing sleep functionality using Controller Area Network (CAN) bus debugger. This required some setup to get ready. The device works through internet; when the IP address of the device is

called it will send a signal to the test device to perform the sleep functionality. The client had written a way to control the device and this method then was implemented into Tau tool. However, one problem faced during this process was the difference in Python versions. The way that the client controls the device was done in later version of Python that Tau used so a way to control this had to be developed as well.

After the device was able to be controlled some other issues were faced. Sometimes the device did not perform the sleep function at all without any reasons as to why it would not perform it. Additionally, while being a high-level problem not really affecting the testing, the screen would turn much dimmer after calling sleep through the device. This made the screen harder to see through the video feed that was recorded during testing though the actions of the device were able to be followed through other means.

With the previous method of doing sleep on the device a crash in the Bluetooth process was seen. This issue persisted even with using the CAN bus debugger device with all the same characteristics of this crash still visible. Otherwise using the device did not add to memory usage or other processes and the test execution time did not go down either.

5.1 Issues observed later into the project

Some more recent issues arose later into the project. One of the biggest new crashes faced was in one of the inhouse system processes. This problem was invisible to the user and did not affect the tests, but it dragged the stability score down. The solution to this was to fix a problem related to Java programming language. Due to this being a low priority problem this problem persisted throughout the latter parts of the project even though the fix would have been relatively simple.

Outside of crashes or ANR:s one of the biggest problems was the device losing connection to the Tau process. Tau installs processes into the test device through which the device is controlled. The main Tau process would start having problems where it would crash continuously after a while of running due to a

java language error. The effect of this problem is that tests would not be able to run and the only way to get the device back online would be to physically reboot the device. In addition, it would cause test cases to fail and would cause stability scores to plummet.

The problematic element of this is that while the issue seemed quite simple to fix this did not appear on other devices and fixing the issue might cause problems on those. Additionally, the software tested had seen its fair share of problems, so it was not completely unreasonable to assume that the issue could be fixed through some other way with the device itself. Different ways to fix this were tried, such as trying to change the USB cables or trying different versions and even computers but nothing seemed to help. When this issue was faced the runs had to be stopped mainly due to the dashboard being filled with crashes in this process. This issue also happened regardless of the connection method since the connection to ADB over ethernet was attempted as well without helping with the issue.

After stopping the run the device could be configured to be back online usually by rebooting the device and trying again until the process would start crashing again. In the end this problem did not get solved and it continued to make some test runs fail. The final solution that came after the project ended was to create a new version of Tau which, in the instance of the process crash, would not try to reinstall the process forcefully to prevent the crashing loop. This would mean that the process would still be faulty and would stop test execution but at least now the run does not need to be stopped if the crash starts to happen.

Another thing that might cause some problems was the hardware part for internet connection. This part would need to be replaced if it caused some problems since some of them would be faulty. There had been some other internet problems before as well which could be fixed if the time on the device was configured manually to the correct time. This fix worked sometimes but towards the end of the project this fix was not used and instead the hardware part was changed. This is since the time fix was quite unreliable and would not work all

the times. Additionally, this fix was difficult to do with remote connection and having the delay from it to deal with.

6 CONCLUSION

Many things had changed, and many things were noticed throughout the project. Firstly, the stability of the software had increased considerably . The biggest issues noticed at the start of the project were solved and some smaller ones had been fixed as well. The main problem with the memory leak was a big find and it was fixed quickly at the start of the project. This is not to say that all problems were solved: some things, while found, did not get solved due to various reasons. Nevertheless, the overall stability of the software saw an increase.

Many of the additional parts, both hardware and software wise, were also given a lot of attention. Some of the problems related to these things were found and general understanding of these elements was also increased. Many of these problems were also fixed in the end even though not all the problems were solved. However, the stability of these elements also increased . The way customer was informed about the problems found was by doing reports and then the customer could use their own issue tracking to inform teams about them.

Overall, the customer seemed happy about the progress made during the project. Despite this, not everything went as planned. One of the requirements of the customer was to have at least 90 test cases; however, in the end this amount was 60. The reason for this was that the test environment was more closed down than expected and many of the potential test applications needing information such as licence plate number made testing these applications impossible. This was not a major issue however it still should be mentioned that this part did not meet the requirements.

On top of this, while the software became much more stable there were many parts which still encountered some instability. Some of the new system components that were also added did not receive as much testing as others. Due to many of these components simulating the real situation this also could cause some issues down the line though this is something that cannot be tested through the means used in this project. Testing in simulated environment does help but it does not guarantee fully that it fixes all problems in non-simulated situation.

There were also many problems encountered that did not relate to the actual system itself. Things such as the testing computers having problems or slow USB speeds did get in the way of some of the test runs. Additionally, there were a few cases of test results having unwanted crashes such as the Tau process that showed up to the test results. In the end while some these problems were present, they did not affect the overall testing too much and sufficient results were achieved.

In the end this project did prove that testing in software development is quite important. The stability of the software at the start was quite poor, with the system having to reboot itself after a while. In the end the problems ended up being smaller issues with some applications being slow. Of course, the stability of the system is not perfect and there is still work to do however from where everything started the improvement is visible.

REFERENCES

1. Charette, N. 2021. How Software I Eating The Car. IEEE Spectrum. Date of retireval 17.4.2023. https://spectrum.ieee.org/software-eating-car

2. Software testing help. 2022. Functional Testing Vs Non-Functional Testing. Date of retrieval 24.1.2023. https://www.softwaretestinghelp.com/functional-testing-vs-non-functional-testing/

3. Geeks for geeks. 2022. Differences between Black Box vs White vs Grey Box Testing. Date of retrieval 2.2.2023. https://www.geeksforgeeks.org/difference-between-black-box-vs-white-vs-grey-box-testing/

4. Piett, S. The different types of software testing. Date of retrieval 2.2.2023. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

5. Gauthier, M. 2022. QOTD: How Far Do You Typically Drive Each Day? Date of retrieval 3.2.2022. https://www.carscoops.com/2022/04/qotd-how-far-do-you-typically-drive-each-day/

6. Java language documentation. Date of retrieval 9.2.2023. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html

INITIATION DOCUMENT OF A BACHELOR'S THESIS

Author _____

Customer _____

Customer's contact person and information _____

_____

Title _____

Description _____

_____

_____

_____

Objectives _____

_____

_____

Target schedule _____

_____

_____

_____

_____

Date and signatures _____