Implementing a UI automated testing for RoKiX Windows GUI

Mikias Gebre

Student number: 72942

Master of Science Technology Thesis

Supervisors: Adj. Prof. Dragos Truscan

Faculty of Science and Engineering

Department of Information Technologies

Åbo Akademi University

February 2021

ABSTRACT

With the growing demands of their software products and short production time, software companies are always trying to shorten the software delivery time. Performing various tests is a critical process to identify the correctness, completeness, and quality of the software in question. Manual testing can be time and money consuming. Automated testing is a solution to this issue, since it will raise the test frequency and give faster and more reliable feedback. Testing at the GUI (Graphical User Interface) level is crucial, since the GUI is where the user interface interacts with the underlying code.

This thesis aims to design and evaluate a UI automated testing system for RoKiX Window GUI software for ROHM Co. Ltd. RoKiX Windows GUI is a WPF (Windows Presentation Foundation) evaluation Kit software used broadly in the company to evaluate various ROHM devices such as sensors and PMICs (Power Management Integrated Circuit). One important aspect missing in RoKiX Windows GUI is a UI automated testing system. Currently, the software has been tested manually by multiple engineers in the company. This task is taking time and money from the company. Automating the testing process will save time and money for the company.

This thesis introduces and evaluates a UI automated testing system to integrate into RoKiX Windows GUI software. Microsoft WinAppDriver (Windows Application Driver) is used as a testing framework with our custom automation script. The primary tool used is Microsoft Visual Studio 2017 and C# as a programming language to write different customs automation test scripts. The results presented in this thesis show that introducing a UI automated testing system brings numerous advantages, such as making the test results faster and more reliable. It can handle repetitive tests that will free up time for engineers to focus on other tasks, saving time, resources, and money.

Keywords: UI automated testing, WinAppDriver, structure-based testing, WPF, user interface

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

GUI	Graphic User Interface
UI	User Interface
WPF	Windows Presentation Foundation
WinAppDriver	Windows Application Driver
DOM	Document Object Model
HTML	HyperText Markup Language
FFT	Fast Fourier Transform
ROI	Return Of Investment

1. Introduction

This thesis is written in collaboration with ROHM CO. Ltd department of the software development center. It aims to design and evaluate a UI automated testing system for RoKiX Windows GUI software.

1.1 Background

Software testing is a necessary process to identify the correctness, completeness, and quality of the software. All testing on the Graphical User Interfaces (GUI) for RoKiX Windows GUI is manually tested. This method of testing is time and money consuming for the company. As a result, performing UI tests on the software is rarely done. ROHM sees great benefits in expanding the automated testing system to include a UI automated testing system, since testing at all levels is an essential aspect of a software development cycle. Thus, automating it will increase the test's frequency while lowering manual testing costs [1].

UI automated testing is a practical technique for automating high levels of test cases in a software system. This implementation aims to expand RoKiX's testing system by introducing a UI automated testing system to work side by side with the manual testing system. ROHM's long-term goal is to reach a high-level automated testing level for RoKiX windows GUI, which means minimal manual testing on the system.

Manually testing the RoKiX Windows GUI software is currently taking three to four engineers several working hours when doing public releases. In contrast, UI automated testing can be run with almost no cost. However, there is an initial cost for the script's development, which is a process that usually takes several working hours. Besides, the script can run several times. Thus, when considering the cost of manual testing and the development cost, the script's reusability should also be considered.

When having public releases, ROHM has a document list of new features added in the latest release. It also has a list of tests to be done and the expected results of the test. This document also includes maintenance costs since it needs to be updated regularly. We will also add the newly created script's maintenance cost as such, the scripts need to be updated again with the document. The scripts may need to update when, for example, there is a change in the UI layout of the software for the new release.

1.2 Scope of the work

This thesis aims to investigate and evaluate the ability to automate UI testing of RoKiX Windows GUI software at ROHM CO. Ltd. By automating UI tests, the testing process can be more effective while running tests more frequently. The automation is done using Microsoft WinAppDriver. Since RoKiX Windows GUI is a WPF application, the number of such free tools on the market is limited. Other tools that could have been used instead of WinAppDriver are FlaUI and Test studio. The three tools are evaluated and further described in Section 4.2.1. The test results are assessed at the end, comparing the tool's time, effort, and practicality.

Chapter 2 discusses the different methods of automated software testing. This chapter also reviews manual testing and test execution and reporting. It also compares each method's advantages and disadvantages. Chapter 3 gives an overview of the RoKiX Windows GUI software. It explains in detail each element of the software as well as its functionality. Chapter 4 discusses the implementation of the UI automated testing system. Chapter 5 presents the result of the case study and offers some future improvements to the testing system.

2. UI automated testing

This chapter introduces and discusses different methods of UI automated software testing. It also covers manual testing, test reporting, and text execution. This chapter aims to give an overview of different testing methods with their advantages and disadvantages.

2.1 Software testing life cycle

2.1.1 Test design

Rafi et al. [5] highlight the importance of a test design. The quality of the software test and the fault detection rate depends on the test design quality. Both manual and automatic testing requires a test design.

Myers et al. [22] distinguish between white-box testing and black-box testing in their research. White box testing is based on an analysis of the internal structure of the software under test. It is not possible to measure line coverage if the source code is not available. Jamil et al. [23] state that white box testing is useful since it tests the software's functionality and its internal structure. The developer is required to know the programming language to design test cases. White box testing or glass box testing can be applied to all testing levels, including integration or system. Myers et al. [22] mention equivalence partitioning, where the goal is to divide the input data of a system into partitions of comparable data. Another method discussed in [22] is boundary-value analysis. This method is usually efficient to test boundary values. This method can be combined with equivalence partitioning. For example, if we test a pizza order text field that accepts values from 1 to 20. Since testing all cases is not feasible, the test cases are divided into manageable portions.

One drawback of equivalence partitioning and boundary value is they do not consider every different combination of input values. If we consider two values, one positive and one negative value inside a text field, the number of all combinations of both fields is extensive. Hence, it is not possible to cover all subsets of cases even if using automated testing. Myers et al. [22] propose cause-effect graphing as a solution. It is a testing technique that illustrates the relationship between an outcome and factors that influence the outcome. The visualization can help in selecting suitable test cases.

Myers et al. [22] present error guessing as an alternative method. Error guessing is when the tester uses his experience to guess the application's defective areas and prepare test cases for those faulty areas. While this testing method is not precise and difficult to use for many test cases, they [22] suggest it can be useful if used with other conventional techniques such as boundary value analysis and equivalence partitioning. Another method close to error guessing is exploratory testing. Exploratory testing is a method where the test cases are not prepared in advance, and the tester checks the system without any specific plan. According to Itkonen and Rautiainen [24], exploratory testing is a creative method of testing that can help the tester discover some unexpected bugs. Whittaker [25] explains the selective testing method. Selective testing method is convenient to find defects in software areas based on typical usage, since it is easier to find faults in less used areas.

2.1.2 Test execution

The test execution phase is a critical phase where all test cases are executed. Berner et al. [26] highlight the importance of this phase and present a case study where the tests are not adequately maintained. In their case study, initially, there are a limited number of test cases continuously maintained with bug fixes within a short period. However, when the number of test cases increases, the number bugs increases as well. On some occasions, those bugs are not fixed due to priorities. This may lead to a situation where the number of broken tests is high and fixing those bugs becomes expensive. Thus, executing some test cases toward the end of the project becomes difficult. Berner et al. [26] specify that the project's end is crucial to run the test cases. It is also a common reason why software projects are delayed. According to [26], 60% to 80% of bugs found during test execution are found during the development of test cases. Only 20% to 40% of bugs are found during repeated execution of tests. As a result, according to [26], it is beneficial to prioritize new test cases rather than maintaining current test cases.

Prioritization of test cases is discussed in Karlson and Radway [27]. As discussed in [26], it is ideal for running test cases as often as possible. In automated testing, it is necessary to limit the repeated number of test cases or limit the number of test cases due to the long execution times. To solve this issue, Karlson and Radway [27] propose a method where each test case is given a priority tag. This tag is calculated based on failure rate, execution time, and frequency of usage. The test cases that have longer execution times have a low priority tag. Hence, these test cases are run less frequently. At the same time, test cases that failed or that are critical features are tested often.

Re-executing failed tests can improve the reliability of test cases, according to Al´egroth et al. [6]. Their study [6] reported that visual GUI testing often fails due to recognition errors. This can happen when an image is not recognized even though it is there. They also notice that some test failures are random, and rerunning the test fixes the issue. In their case study, they would rerun a failed test three times.

The quality of the test execution depends on the test plan quality. When a test plan is adequately prepared, the test execution is easily implemented [28]. The test plan is continuously improved with the system.

2.2 Type of testing

2.2.1 Manual Testing

Manual testing is a method of testing where developers manually execute test cases. It is still the most widely used method of testing in the software industry [19]. The manual testing structure can follow a specific predefined test case, or it can be investigative testing. Investigative testing means that the developer has the freedom to test the software without any test cases. Structure test cases can follow detailed instructions. These instructions can include the testing phase, test case execution order, and the expected outcome. Al´egroth et al. [6] present an automated test suite based on the manual test cases. Their manual test cases were used as a specification for automated cases.

According to [21], there are six manual testing procedures, as seen in Figure 1. The first procedure is the requirement analysis. The primary task of the requirement analysis are to review the test case, such as product risk analysis, design specification, and design requirements. Other tasks in this phase include identifying test conditions and designing the environment set up. The test planning involves preparing a document that contains the test condition and objectives. This document determines test resources such as the test environment. The tests are created and executed in the test case creation and execution phase, respectively. Defect logging is a process where the defect found or customer's feedback is recorded. The defect fix & Re-verification is when a developer fixes the reported bug and performs a retest on the changed code.

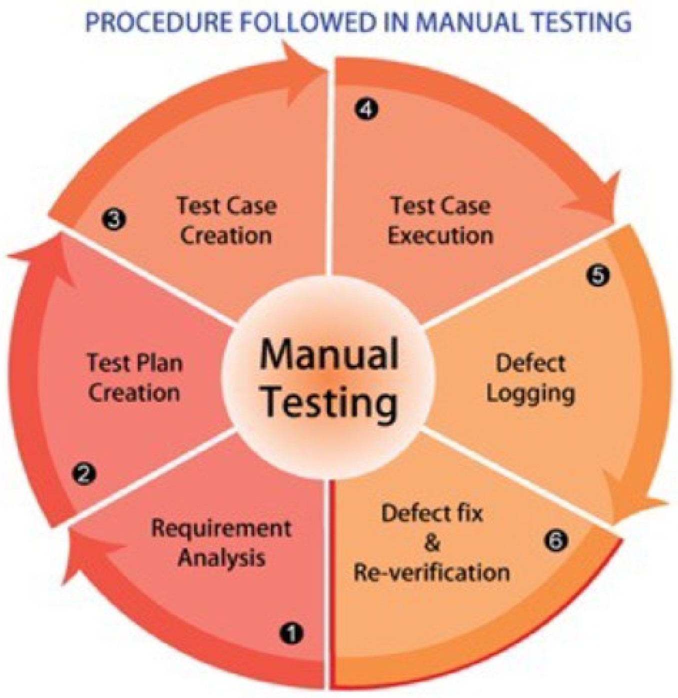


Figure 1: Procedure followed in manual testing [21].

Rafi et al. [5] review in their literature 25 papers regarding the benefits and limitations of automated testing. They did a survey collecting developers' opinions on the finding of their literature review. Their study did not review in detail any specific automated testing method. Instead, they compared, in general, automated testing to manual testing. Some advantages in using manual testing mentioned in the study are reliability of test and test coverage, improved testing quality, increased number of defects detected, reusability of test cases, and lower cost in testing. In another study, Al´egroth et al. [6] compared manual testing to Visual GUI testing. They concluded that manual testing is more expensive to maintain. Also, they pointed out that manual testing is time-consuming since the developer needs to run test cases multiple times. The quality of manual testing depends on the tester's motivation for finding defects in the software. If the tester becomes negligent, it can reduce the quality of the test. The automated tests used in [6] were able to detect more bugs than manual tests. However, the practitioners' opinions were divided equally on whether automated testing can increase fault detection. One participant stated that introducing automation to the testing system does not increase the defect detection rate. He argues that the amount of defect detection depends solely on the quality of the test cases. However, most practitioners agree there are other substantial benefits when using automated testing.

The primary limitations in automated testing, according to Rafi et al. [5], are difficulties in maintenance, high initial cost, false expectations, lack of skilled personnel, lack of exploratory testing, and inappropriate testing strategies. Also, Al´egroth et al. [6] state that it is not feasible and possible to automate an entire testing system. They argue that automated tests can only find bugs explicitly asserted in test cases, unlike manual tests. Manual testing can sometimes be the only method of testing the software. For example, if the software requires a lot of domain knowledge, then automated testing would be difficult to apply to this kind of software. In a survey conducted in [5], only 6% of the participants believe that software testing systems can be fully automated. Most of the participants also agree that there are some limitations to automated testing. About 45% of the participants believe that the testing tools currently in the market are inadequate. The view on tools is divided because the developers have different testing requirements, so they use other testing tools.

There is no definite answer to the question: "Is manual testing dying ?" The debate in the software community regarding the future of manual testing is never-ending and ongoing [20]. Some argue that manual testing is dying, and those working as manual testers don't have a viable future career. Others argue that using "dying" in this argument is extreme. It does not take into consideration the numerous advantages that manual testing brings that automated testing does not. A human can also execute Emotionally-driven test cases. It can position a developer to test an application from a user point of view. Hence, it can give the developer the chance to test the usability and the software user experience. Ultimately, test automation can only replace tasks, not testers[21]. While there is room for improvement in the automation testing tools, it can facilitate and relieve some burdens with testing software manually. Automated testing cannot replace all aspects of testing yet. Humans will always be part of the testing process and add different values to the software's quality.

2.2.2 UI automated testing

User interface automated testing is not a new concept. Kepple [4] introduced this concept in 1994. In his journal, Kepple discusses UI testing with capture and replay tests. UI automated testing has improved during the last twenty years. However, it still has issues and drawbacks. The high initial overhead cost to build UI automation and the lack of skilled people are some of the problems identified. Rafi et al. [5] and Al´egroth et al. [6] discuss in their paper that 45 % of the practitioners think their automation tool is high-priced and believe that the tool is not easy to learn. Therefore, manual testing is still frequently used in the software industry. Nevertheless, automated UI testing is becoming a common practice in the industry. In this chapter, I will compare and study various methods of UI automated testing. Automating all tests is not feasible. Automated testing cannot replace manual testing but should rather be a compliment. [7] I will present different scenarios where manual testing would be better to use than UI automated testing. The first part of this chapter covers various methods of UI automated testing. This part answers which method is cost-effective.

There are various methods for UI automated testing. Leotta et al. [8] give one classification of UI automated testing (Figure 2). Leotta et al. [8] paper is based on web applications. However, the same classification can be applied to Windows Desktop applications such as WPF (Windows Presentation Framework) and UPF (Universal Presentation Framework)). The first separation in UI testing based on Leotta et al. is between *Automated testing* and *Manual testing*. *Manual testing* is a method of testing where developers manually execute test cases. *Capture and replay testing* are also related to Manual testing indirectly. This thesis focuses on automatic testing. Furthermore, *Automated testing* is divided into *programmable testing*, *model-based testing*, and *capture and repay testing*. Each method is discussed further with examples in different sections of this chapter.

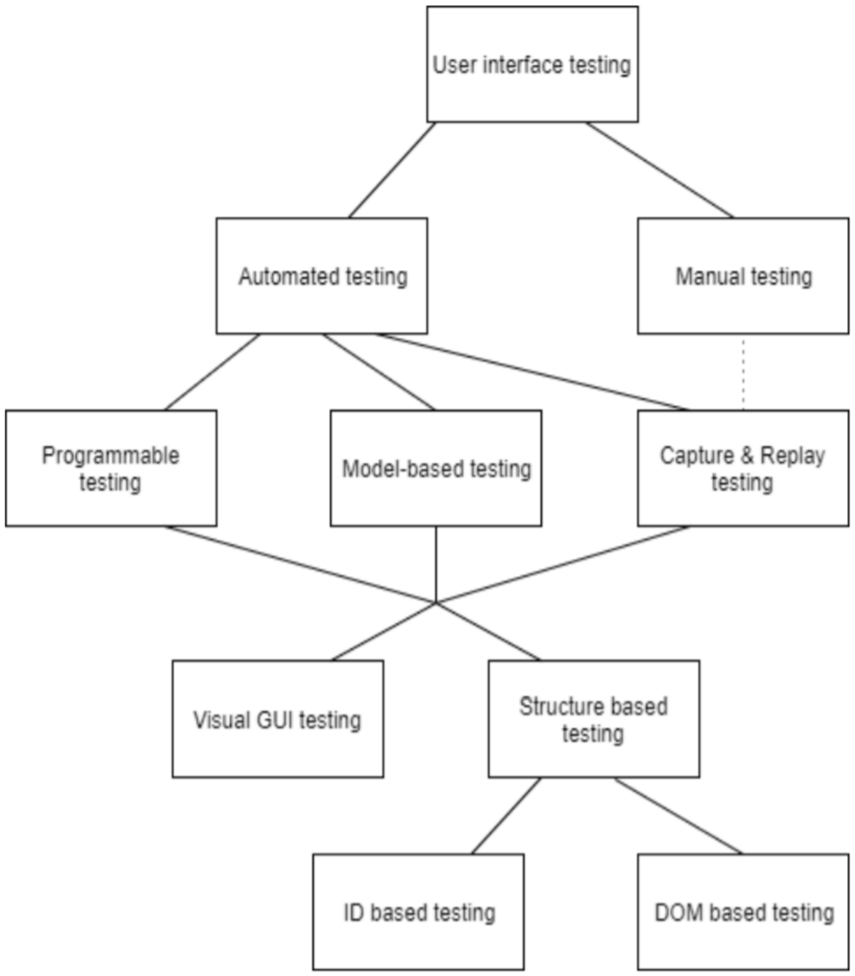


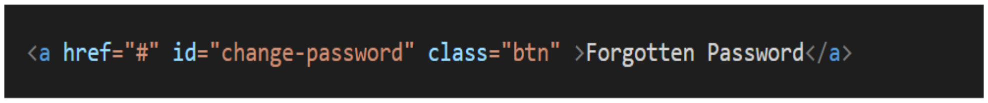
Figure 2: A classification of test methods based on Leotta et al. [32].

In *programmable testing*, the developer writes a script that executes some tests. In *capture & replay testing*, the developer can run an application and record UI interaction between the user and the tested application using a capture tool. The recorded step can be played numerous times. In *model-based* testing, the developer will create a test script that will run against a predicted model. A model, in this case, is a description of a system's behavior. All three methods (*programmable testing*, *Model-based testing*, and *capture & replay testing*) can be categorized into visual GUI testing or structure-based testing. Also, *structure-based testing* can be categorized into *ID-based test*ing or *DOM-based testing*.

2.3 UI automated testing

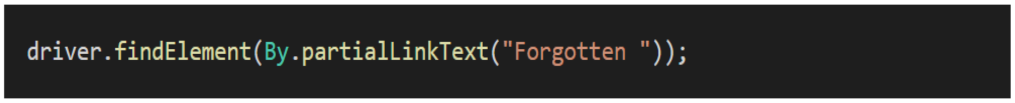
2.3.1 Structure-Based Testing

Structure-based testing is a UI automated testing method where the developer writes test scripts to run the UI automated tests. Usually, in structure-based testing, the UI elements are detected in the script by their element ID. Structure-based testing is broadly used in several web and desktop application testing frameworks such as selenium and Appium. For example, for testing a web application, the needed element ID can be found inside the DOM.



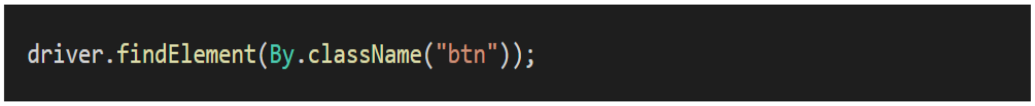
```
<a href="#" id="change-password" class="btn" >Forgotten Password</a>
```

Figure 3: An example code of HTML code.



```
driver.findElement(By.partialLinkText("Forgotten "));
```

Figure 4: An example of selenium code finding element by link Text.

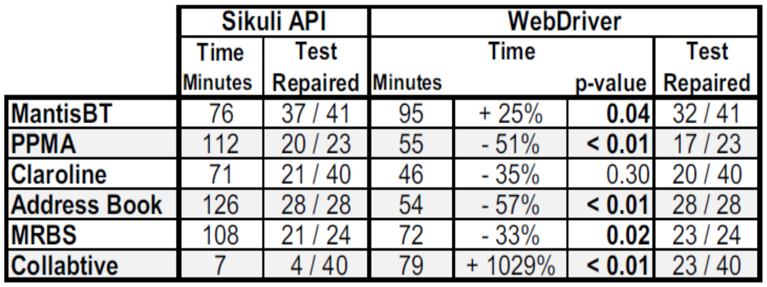


```
driver.findElement(By.className("btn"));
```

Figure 5: An example of selenium code finding element by class name.

Figure 3-5 illustrate the general concept of structure-based testing. Figure 3 shows an HTML hyperlink with an ID and a class name. The ID should be unique to avoid ambiguous reference calls and distinguish the UI elements. In Figure 3 case, the element ID that can be used is the ID or the class name. Figures 4 and 5 demonstrate different methods used in the Selenium script to interact with an HTML code. It would not be possible to interact with different HTML elements if the ID would not be defined. The two methods find the HTML link using the partialLinkText method and secondly using the class name attribute. Both methods yield the same result.

Leotta et al. [8] selected six open-source web applications from SourceForge.net to compare different DOM (Document Object Model) approaches to visual recognition. They choose these six web applications based on their release date and their popularity. Selenium WebDriver is a tool using the DOM approach that identifies and recognizes its element by ID. Selenium also uses LinkText, CSS, and Xpath when locating different HTML elements. Sikuli is a tool using a visual approach. Both tools have at least two different versions. This made it possible to develop test cases with two different versions. Both applications were hosted on a local network, and the tests were evaluated based on the number of locators needed. The other criteria for the tests are the execution time, maintenance costs, development effort, and test robustness.

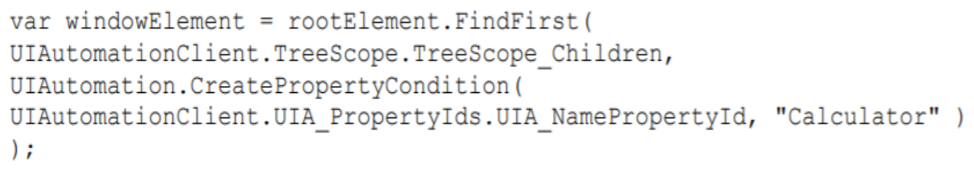


	Sikuli API		WebDriver			
	Time	Test	Time			Test
	Minutes	Repaired	Minutes		p-value	Repaired
MantisBT	76	37 / 41	95	+ 25%	**0.04**	32 / 41
PPMA	112	20 / 23	55	- 51%	**< 0.01**	17 / 23
Claroline	71	21 / 40	46	- 35%	0.30	20 / 40
Address Book	126	28 / 28	54	- 57%	**< 0.01**	28 / 28
MRBS	108	21 / 24	72	- 33%	**0.02**	23 / 24
Collabtive	7	4 / 40	79	+ 1029%	**< 0.01**	23 / 40

Figure 6: Test suite execution [8].

Figure 6 shows Letta. et al. [8] test suite execution result. The table contains the mean execution time, the standard deviation, the difference in percentage between the time required by the Sikuli test suites and the web driver test suites, and the number of tests repaired. The result shows that the execution time to perform Sikuli test suites takes more time than the WebDriver test suites for all six web applications in the study. Sikuli's execution time is higher than WebDriver because executing an image recognition algorithm takes much more time than the DOM approach since the former requires more computation resources.

The same concept can be taken from web application frameworks and can be used for Windows applications. Lehtinen [12] presents Microsoft's UI automation tool framework called TestStack White. This framework utilizes the Component Object Model (COM) and interfaces. It is designed for C/C++ developers and supports all Windows platforms (WPF, Win32, and WinForms). Like selenium, TestStack White programmatically queries UI elements by their ID. Various UI inspection tools can identify these element IDs. Windows application UI elements are arranged in a hierarchy tree. Figure 7 presents an example of how to reference a UI element in TestStack White. The "FindFirst" method returns the first element of the tree collection that matches the provided search criteria. In this example, the requirement here is the UI element with the ID "calculator." Like Selenium, TestStack White also allows the selection of UI elements by their ID or element type.



```
var windowElement = rootElement.FindFirst(
UIAutomationClient.TreeScope.TreeScope_Children,
UIAutomation.CreatePropertyCondition(
UIAutomationClient.UIA_PropertyIds.UIA_NamePropertyId, "Calculator" )
);
```

Figure 7: TestStack White query element by ID [12].

2.3.2 Capture & Replay Testing

"GUI capture & replay" tools have been developed as a mechanism for testing the correctness of interactive applications with graphical user interfaces." [9] Contrary to programmable testing, this method of testing does not require any programming skills. Using a capture & replay tool, a user can run the software to record the entire interactive session. The tool records all the user's interactive steps, such as mouse movement, click, and drag. At the end of the recording, all the user steps will be logged into a file. The testing software can automatically replay the same interactive steps by using the file any number of times without requiring any human interaction.

“GUI capture & replay tool” primary purpose is not to record the entire interactive session. Its primary goal is to record simple interactions such as opening the “File open” menu and verify that the action indeed opens the file menu. Jovic et al. [9] discuss in their paper if the GUI capture & replay tool can be used extensively and record the entire interactive sessions with complex real-world applications. Their studies further examine if the GUI capture & replay tool gives accurate results if used extensively. They evaluated different capture and replay tools and examined their ability to record and replay interactive sessions. The result of their studies shows that various tools (Abbot, Jacareto, Pounder, Marathon, and JFCUnit) are unable to capture realistic interactions with real-world applications.

Furthermore, Leotta et al. [8] compare the cost of programmable tests with capture and replay tool testing. They found out that programmable tests have a higher cost to create initially than the capture and replay tools since the latter generates the code automatically based on user action. However, their study notes that capture and replay test’s lack of code reuse makes them more expensive to maintain in the long term. According to their studies, a programmable test’s total cost is considerably lower if new versions are released regularly. It also highlights that programmable tests give more opportunities for building tests and parameterizing them. [8]

2.3.3 Visual GUI Testing

“Visual GUI testing is a tool-driven test technique where image recognition is used to interact with, and assert, a system’s behavior through its pictorial GUI.”[12] The visual testing method for locating UI elements differs from structure-based testing because it is independent of the software’s source code under test. Visual GUI Testing (VGT) core is image recognition, used to analyze and interact with the bitmap layer of a system UI element. VGT has been improved throughout the years with perceived higher flexibility and robustness. Using image recognition combined with a test script, VGT tools can imitate user interaction user behavior on any GUI based system. The VGT is entirely independent of the system. The system can be written in any programming language or can run in any operating system. Some examples of tools using VGT methods are Sikuli, JAutomate, and TestComplete. Contrary to structure-based testing, VGT would not locate UI elements by their ID, text, or other DOM feature. Instead, the tester would take a screenshot of the element, and then the testing tool would search for an object identical to the screenshot on the screen.

In another study, Al´egroth et al. [6] presented a case study in Saab using the VGT tool to test two different projects independent of each other. In the study, the development team used Sikuli and another unnamed tool. In the testing document, they had 67 manual tests conducted. Their main objective was to automate those 67 manual tests. The study reports that the developers find the automated test practical. However, they also noted that automated testing using Sikuli has numerous problems and reported 58 issues when using the tool. The issues can be categorized into 26 different categories. Some of the issues reported in this study are test script failure, synchronization issues, and image recognition errors due to it being performance intensive. They also present some solutions to these issues. The first solution is to ensure the test scripts are robust by having multiple exception handling.

One of the major issues reported for the Sikuli is the precision of the image recognition algorithm. It does not work well in all test cases. Sikuli's image recognition does not require the screen's element to be identified. Instead, it accepts any image that has some similarities. The level of similarity is calculated based on pixels. Some elements might not be recognized in many situations where a human would recognize them easily. Al´egroth et al. [6] observed in their study that failures occur randomly. When retesting these failures, they sometimes succeed in recognizing an element, and sometimes they fail. One possible solution to this problem is to use different images. Using the same image can also work as this sometimes solves the issue.
Leotta et al. [8] report the synchronization issue when using VGT tools. This problem occurs when the test has to wait for the elements to appear on the screen to perform the

test. This can be solved by adding a waiting period or a delay. As such, the test execution time may get lower. Another issue reported in the same study was that the Sikuli documentation is insufficient. This makes debugging problems challenging.

Karlson and Radway [13] discussed a Visual GUI testing implementation for a website. The website is an e-commerce site where customers can search for product information and purchase them. Similar to Al´egroth et al. [6] and Leotta et al. [8], they also identified the execution time issue of the Visual GUI testing. The significant advantage of VGT is that it is easy to create and maintain tests because they are independent of the source code. The visual GUI testing approach makes testing intuitive and easier to understand for developers with an inadequate understanding of the source code. Having the test system independent of the source code can also make the tests more robust. Visual GUI testing is a suitable method for regression testing since new features are not added frequently.
Alégroth et al. [12] describe a technique to test a windows application using VGT (Visual GUI Testing). This tool uses image recognition to interact with and assert a system's behavior through its pictorial GUI as shown to the user in an application. Alégroth et al. [12] objective is to evaluate the long-term use of VGT at Spotify's Windows application. Besides, their paper presents the challenges and benefits of using this tool to integrate into Spotify. The second objective of this paper is to examine other alternative techniques that can be used at Spotify. Based on their study, it was concluded that using VGT for Spotify can be beneficial in the long term. However, they also note that there are challenges when using this tool. The challenges presented are the high maintenance costs of testing and the limited applicability of mobile applications. They also offered an alternative means to VGT called the Test interface. This tool has several benefits compared to VGT, including lower maintenance cost and higher use flexibility.

Also, they present a case where visual GUI testing is used in the long-term. They discuss the use of the Sikuli tool for GUI testing at Spotify for over several years. To conduct and collect data, they interviewed five testers in the company. Sikuli was introduced in 2011 by combining Visual GUI testing and model-based testing. The model-based test cases were created using the Graphwalker tool, discussed in detail in section 2.3.4. In a survey conducted in the research, most employees using Sikuli at Spotify did consider Sikuli reliable. 99.9% of the time, Sikuli image recognition correctly identified the image. The developers also added a significant advantage when using Sikuli is having the source code and the testing system independent. As such, they were able to test the software with different platforms and operating systems. They concluded that most of the developers were satisfied with Sikuli.

Later, Spotify decided to replace Sikuli with another testing tool due to some visual GUI testing drawbacks. The introduction of dynamic elements to the UI caused several issues with Sikuli since it expected a result based on a static picture. Every test conducted with dynamic elements with Sikuli failed. As a result, they decided to test only the static UI elements, which would not change at runtime. Another issue reported is the high maintenance cost. One reason for the high maintenance cost is updating the Sikuli tool's picture when the user interface is changed. Also, the software in the test has a slightly different appearance in different operating systems. This will result in having different versions of test cases for different operating systems. Furthermore, Sikuli does not support testing on mobile devices.

2.3.4 Model-Based Testing

Until now, we have disused testing methods that require interaction with the UI elements with an identifier. The methods discussed earlier (structure-based testing and visual GUI testing) need the developer to declare the elements before testing them explicitly. In Model-Based Testing (MBT), the test cases are executed based on a given model. This model is an abstract and a partial presentation that describes the software's expected behavior under test.[15] MBT uses different system characteristics under test to generate automated testing for different parts of the software systems.

Utting et al. [16] argue that there are several advantages in using Model-Based Testing. Some benefits include low maintenance cost, enhancing test quality, and automated test design. Figure 8 presents the process of model-based testing. The model is constructed based on the gathered requirements of the system. Additionally, test selection criteria are also created based on needs. They are used to select test cases that detect errors, faults, or failures. “Test case specifications are built from test selection criteria used with system models to generate actual test cases.” [16] The test results are analyzed by test verdict after executing the test cases of the system under test.

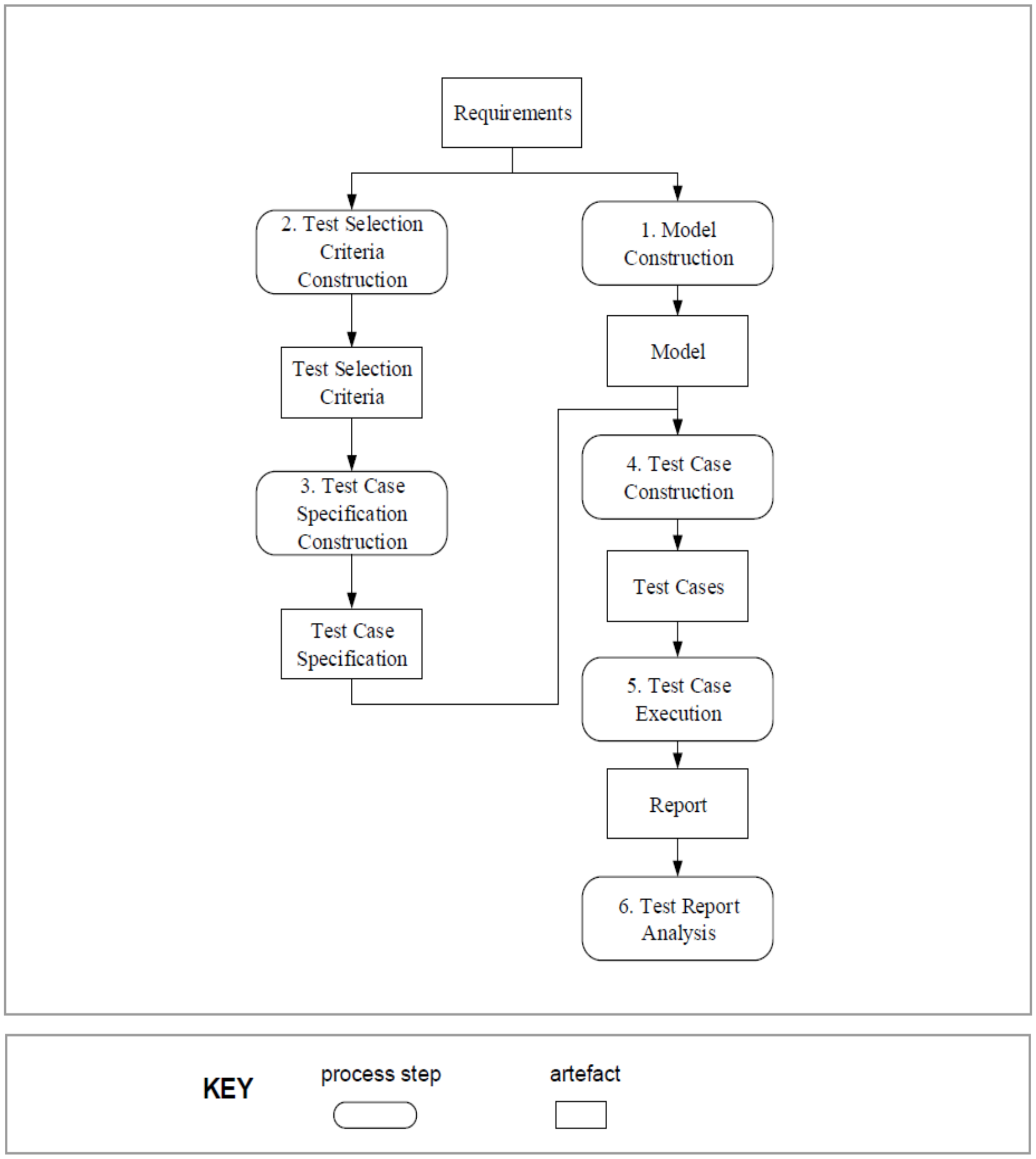


Figure 8: Process of Model-Based Testing [16].

Amalfitano et al. [17] present MobiGUItar (Mobile GUI Testing Framework), a testing tool that works with Android applications using model-based testing. They have used it with four different mobile applications. MobiGUItar builds a scalable state machine in the user interface by trying different actions in the user interface. MobiGUItar's state machine model with test coverage provides a way to generate test cases. Amalfitano et al. [17] argue that MobiGUItar managed to report some defect related to the application crashing. However, the tool cannot detect all possible defects. It can only find fatal defects that mean defects that will crash the application.

Neto et al. [18] have made a systematic review on model-based testing by analyzing 78 research papers. MBT approaches are usually separated from the software development process. The models used for testing are defined exclusively by an MBT approach. The MBT method of testing does not provide a mechanism to test NFR (Non-Functional Requirement) such as usability, security, and user interaction. Using an MBT approach requires advanced knowledge of the modeling language, criteria, and testing coverage. They also argue that most MBT approaches are not evaluated extensively and transferred to the industrial environment. Without an empirical knowledge of the effort, cost, and quality, it is challenging to generalize and apply the MBT approach to the industry level.

2.3.5 Comparison of the different testing method

In the previous sections, various testing tools and techniques have been discussed. This section compares multiple methods of testing and evaluates their advantages and disadvantages. The result of the comparison summarizes in Table 1. Automated testing is compared to manual testing. Then capture & replay and programmable testing are compared. Finally, test prioritization, test case design, execution, and test prioritization are discussed.

Table 1: Advantages and disadvantages of the different testing methods.

Method	**Advantages**	**Disadvantages**
Manual testing	Low initial development cost [5] allows the tester to perform Adhoc testing Enable creative testing	Time-consuming High cost over time [5] Test result depends on the ability of the tester.
Capture & Replay testing	Low initial cost No programming skill required [8] Help the developer understand the source code better [29]	Low Return On Investment (ROI) [29] High maintenance cost [8] Depend on the precise placement of UI elements [29]
Structure-based testing	Robust and low maintenance cost [8] Require more attention to the source code and the internal implementation [30]	High initial cost [8] In-depth knowledge of the programming skill is required [30] Time and energy consuming [30]
Visual GUI testing	VGT is independent of the source code [12] More intuitive for developers [6]	Image recognition accuracy issue [6] High initial development cost [8] High maintenance cost in some cases [6]
Model-based testing	It helps create more quality software by getting developers to think about the software model [31] Reduce test suite maintenance [31] Flexible to generate multiple tests using different algorithm [31]	The learning curve is steep [31] Difficult and expensive to create a suitable model [17]

The most unchallenging automated testing method to start with is capture & replay testing since it has a lower development cost and does not require any programming skills. Leotta et al.[8] note that capture & replay testing maintenance cost is high. This is because it is not feasible to reuse the code in capture & replay testing. Small changes in the software under test might require recording all the tests again. High maintenance code will not be an issue if the software under test will not require code change often.

According to Leotta et al. [8], in programmable testing, Visual GUI testing and structure-based testing are the two most used testing methods. They [8] note that DOM-based testing and structure-based ID testing requires less effort and time to develop. In 4 of 6 cases, maintenance costs were lower. According to [30], structure-based testing forces the developers to investigate better and understand the application's design. This can make the structure of the application robust.

"Structure-based testing methods have various locators that can be used" [32]. According to Leotta et al. [8], ID-based locators are easily maintained compared to XPath or LinkText based locators. ID-based locators are more precise in locating UI elements. Also, ID-based locators can be used with ID created dynamically. Also, ID-based locators do not change frequently.

Model-based testing is not an alternative to structure-based testing, programmable testing, or visual GUI testing. Instead, it is intended to be used with the method mentioned above. Alégroth and Feldt [12] present a successful case study where model-based testing is used with Visual GUI testing. Sikuli is used to locate UI elements by ID with the Graphwalker tool to generate Graph based on the model. As such, a developer with no programming skills can understand the model.

Whittaker [25] discuss different test coverage methods, such as boundary-value analysis and equivalence partitioning. The test coverage methods mentioned above can be integrated with model-based testing, and it can be analyzed against a model. Going through all the states in the model would give full test coverage of the functionalities. Building a good model does not automatically guarantee a full test coverage of the functionalities.

Figure 9 shows the selected method for implementing UI automated testing for RoKiX Windows GUI. Model-based testing and capture & replay testing were not used in this project. The capture & replay method was not used due to high maintenance costs. Model-based testing was not used because of time constraints.

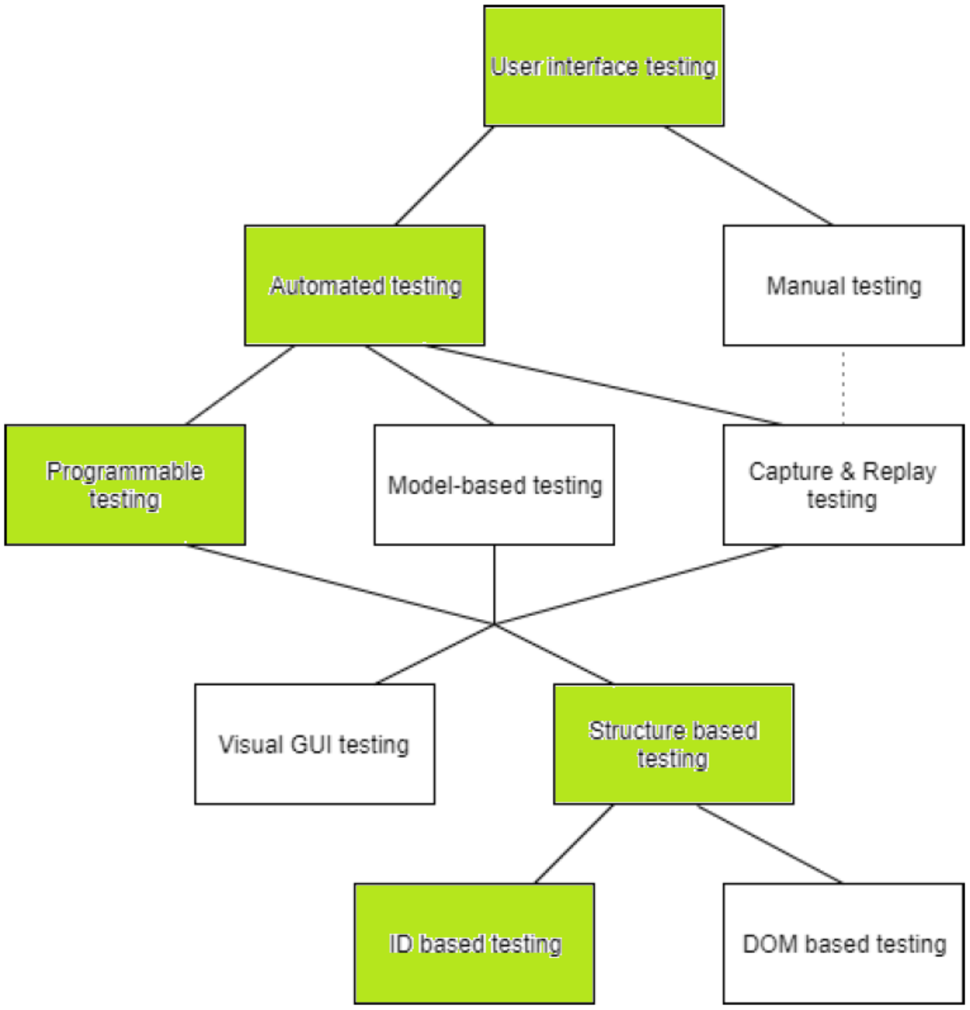


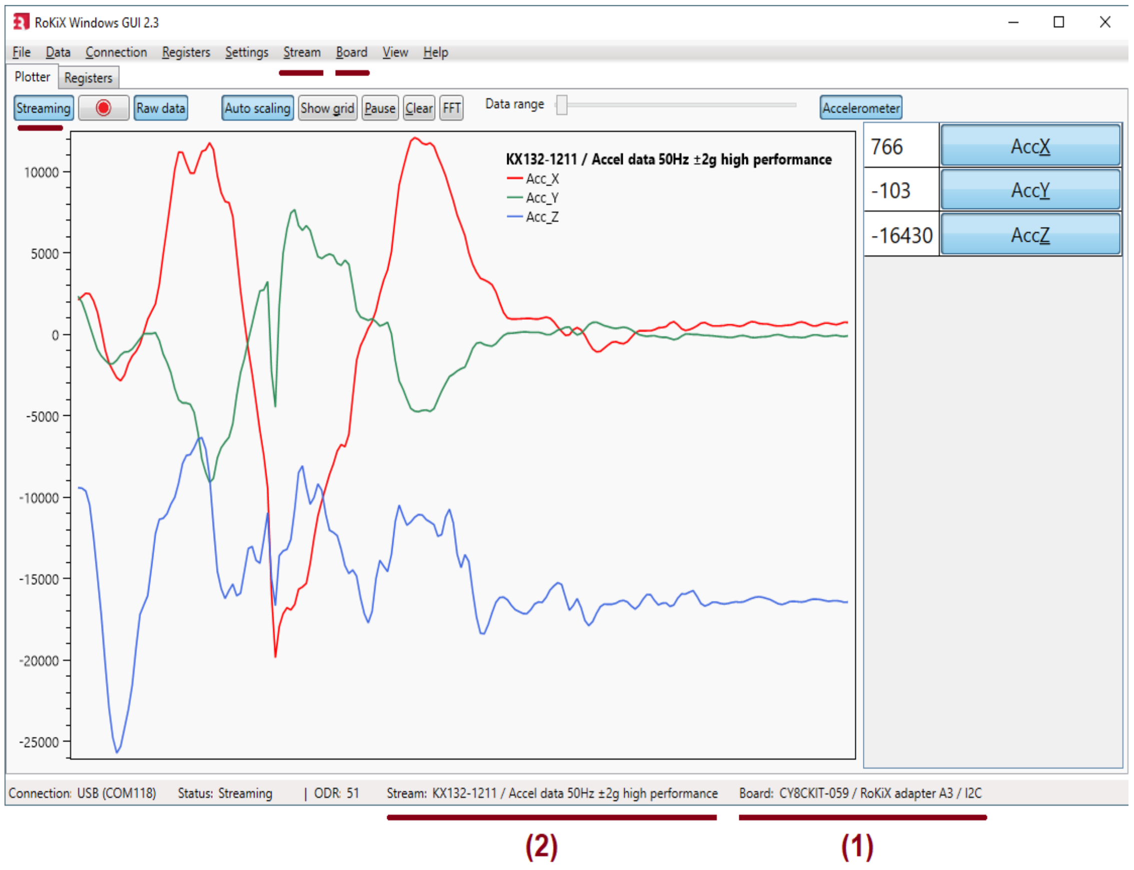
Figure 9: The selected method for the case study.

3. RoKiX Windows GUI

RoKiX Windows GUI is part of the RoKiX IoT platform Client software, which provides an easy-to-use graphical user interface demonstrating high-level device offering and features. [2][3]. The RoKiX IoT platform is an IoT application that can visualize, collect and process sensor data via the RoKiX sensor Node. RoKiX Windows GUI helps evaluate different ROHM devices such as sensors, LEDs, and PMICs. It offers three significant features: a visual display of real-time device data, the ability to record device data into a file and read or write registers into the device. It is compatible with Windows OS versions 7, 8, and 10.

3.1 RoKiX Windows GUI plotter Tab

RoKiX Windows GUI contains two tabs: The plotter and the Registers tab. The plotter is used to show real-time data from the connected device, as shown in Figure 10. The plotter in Figure 10 is displaying real-time output from KX132-1211(tri-axis 16-bit accelerometers). To stream data, users should configure the device accordingly by choosing the right board configuration from the Board menu and the correct stream configuration from the Stream menu from the Windows menu bar. The "Sub channel view" is enabled by default on startup. This view can be seen on the right side of the plotter. It is used to show the digital output of data.


RoKiX Windows GUI 2.3
File Data Connection Registers Settings Stream Board View Help
Plotter Registers
Streaming Raw data Auto scaling Show grid Pause Clear FFT Data range Accelerometer
KX132-1211 / Accel data 50Hz ±2g high performance
Acc_X
Acc_Y
Acc_Z
766 AccX
-103 AccY
-16430 AccZ
(2)
(1)

Connection: USB (COM118) Status: Streaming | ODR: 51 Stream: KX132-1211 / Accel data 50Hz ±2g high performance Board: CY8CKIT-059 / RoKiX adapter A3 / I2C

Figure 10: RoKiX Windows GUI Plotter Tab [2].

The plotter can be controlled using different toggle buttons positioned on top of the plotter. (Figure 11). The Streaming toggle button is used to start/stop streaming. Data logging can be enabled and disabled using the toggle button with the red circle icon. The red circle starts blinking when the logging is enabled. Using the “Auto scaling,” the plotter will auto-scale the minimum and maximum values in the y-axis according to the device data. The Show grid button enables data grid lines. As the name suggests, the pause and clear toggle button pause and clear all plotted data points. FFT turns on the FFT (Fast Fourier Transform) functionality of the plotter. This feature can be used to show frequency data. The data range is a slider bar that adjusts the number of data points shown in the plotter.

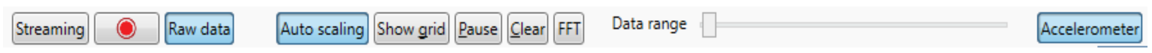
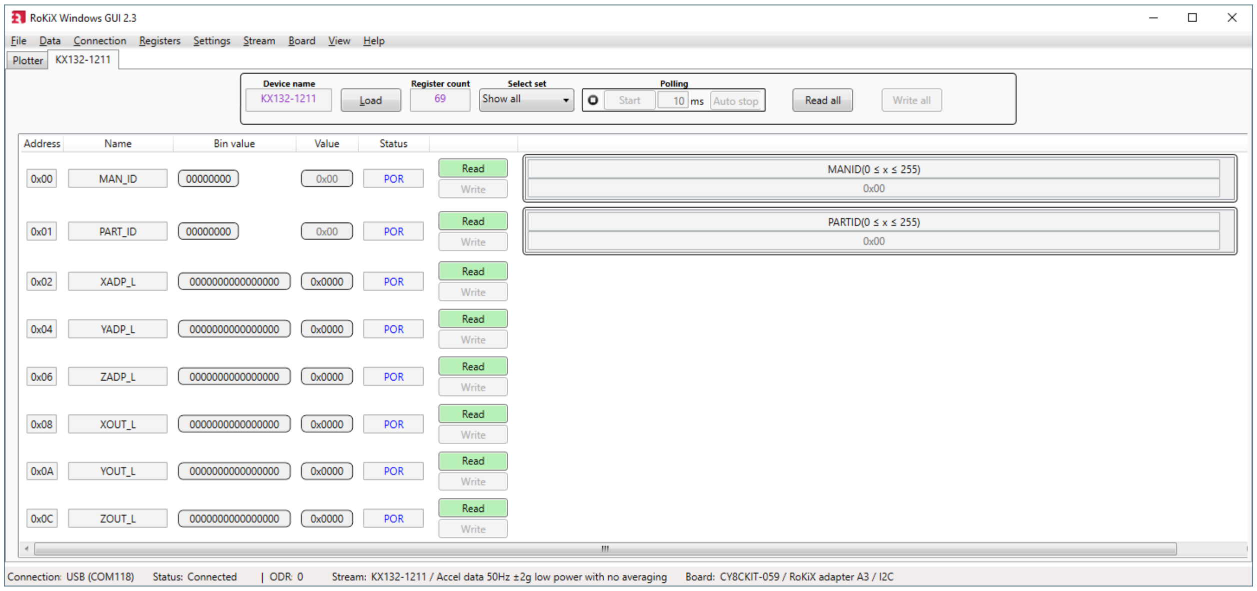


Figure 11: RoKiX Windows GUI plotter settings [2].

3.2 RoKiX Windows GUI Registers tab

The register editor tab is used for reading and writing device register values. (Figure 12). Using the Read and write button, users can read and write values to specific registers. Also, it is possible to read all the registers using the Read All button.



Device name: KX132-1211 | Register count: 69 | Select set: Show all | Polling: 10 ms

Address	Name	Bin value	Value	Status
0x00	MAN_ID	00000000	0x00	POR
0x01	PART_ID	00000000	0x00	POR
0x02	XADP_L	0000000000000000	0x0000	POR
0x04	YADP_L	0000000000000000	0x0000	POR
0x06	ZADP_L	0000000000000000	0x0000	POR
0x08	XOUT_L	0000000000000000	0x0000	POR
0x0A	YOUT_L	0000000000000000	0x0000	POR
0x0C	ZOUT_L	0000000000000000	0x0000	POR

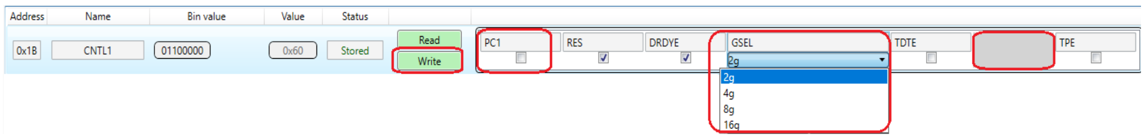
MANID(0 ≤ x ≤ 255) 0x00

PARTID(0 ≤ x ≤ 255) 0x00

Connection: USB (COM118) Status: Connected | ODR: 0 Stream: KX132-1211 / Accel data 50Hz ±2g low power with no averaging Board: CY8CKIT-059 / RoKiX adapter A3 / I2C

Figure 12: RoKiX Windows GUI register editor [2].

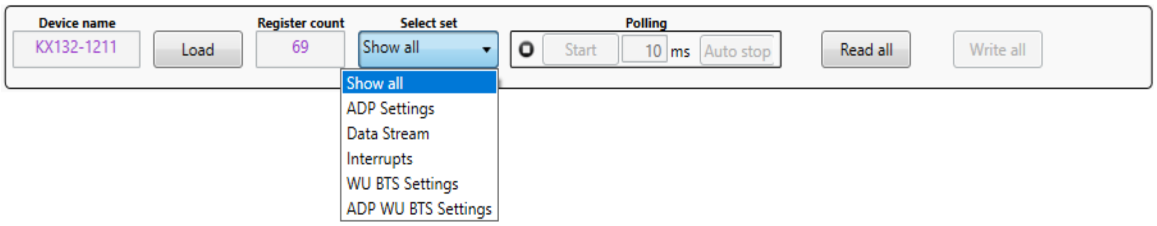
This graphical user interface provides an intuitive approach to update the values of certain bits and entire registers. (Figure 13). When a single bit defines a particular function of a specific register, the bit's value can be changed by checking or unchecking the checkbox (For example, PC1 bit in Figure 13). Some bits are grouped if they make up one setting and have a predefined function for each combination of bits (for example, the GSEL in Figure 13) [2]. If the bit is grayed out, it means it is a reserved bit, and thus, the register editor does not provide a way to modify it to avoid unexpected behavior. [2]



Address	Name	Bin value	Value	Status
0x1B	CNTL1	01100000	0x60	Stored

Figure 13: CNTL1 register of KX132-1211[2].

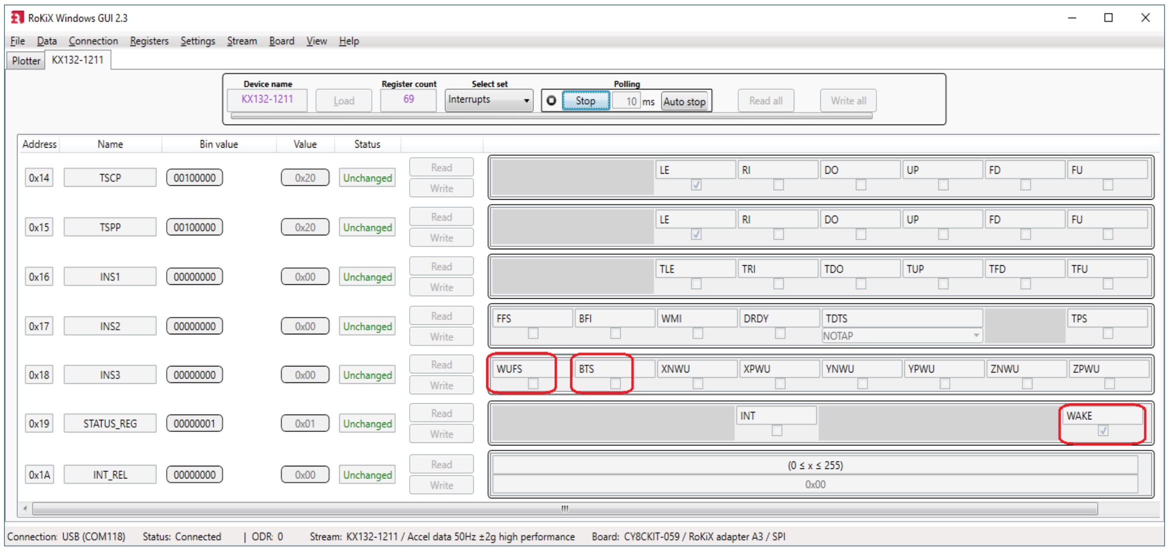
The register sets functionality allows users to see registers with common functionality. For example, when a user selects ADP settings, the register editor will display only registers related to ADP settings. (Figure 14)



Device name: KX132-1211
Register count: 69
Select set: Show all
Polling: 10 ms

Figure 14: Select set Drop Down Menu [2].

The register polling is a way to monitor the values of the register sets continuously. The polling functionality can be started and stopped by pressing the Start and Stop toggle button. (Figure 15)



Device name: KX132-1211 | Register count: 69 | Select set: Interrupts | Polling: 10 ms

Address	Name	Bin value	Value	Status								
0x14	TSCP	00100000	0x20	Unchanged			LE ✓	RI	DO	UP	FD	FU
0x15	TSPP	00100000	0x20	Unchanged			LE ✓	RI	DO	UP	FD	FU
0x16	INS1	00000000	0x00	Unchanged			TLE	TRI	TDO	TUP	TFD	TFU
0x17	INS2	00000000	0x00	Unchanged	FFS	BFI	WMI	DRDY	TDTS: NOTAP			TPS
0x18	INS3	00000000	0x00	Unchanged	WUFS	BTS	XNWU	XPWU	YNWU	YPWU	ZNWU	ZPWU
0x19	STATUS_REG	00000001	0x01	Unchanged				INT				WAKE ✓
0x1A	INT_REL	00000000	0x00	Unchanged	(0 ≤ x ≤ 255) 0x00							

Connection: USB (COM118) Status: Connected | ODR: 0 Stream: KX132-1211 / Accel data 50Hz ±2g high performance Board: CY8CKIT-059 / RoKiX adapter A3 / SPI

Figure 15 : Wake-Up / Back-to-Sleep Interrupt Detection Register Set [2].

3.3 RoKiX Windows GUI Menu Bar

The RoKiX Windows GUI menu bar provides settings and options for users to change. The File menu contains only the option to exit the application by selecting "Exit." (Figure 16)

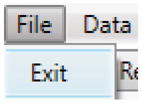


Figure 16: RoKiX Windows GUI File menu bar [2].

The streaming menu is used for enabling or disabling device data streaming. The Logging menu is used to enable or disable logging. (Figure 17)

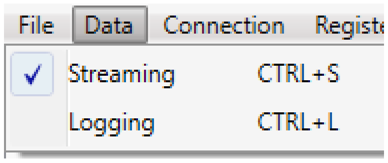


Figure 17: RoKiX Windows GUI Data menu bar [2].

The connection menu is used to change the COM connection type. The RoKiX Windows GUI is using USB COM communication by default. (Figure 18)

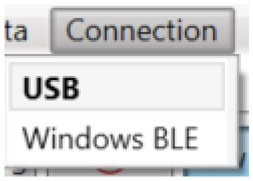


Figure 18: RoKiX Windows GUI Connection menu [2].

The register menu allows users to load the device's register definition file from the "Register" tab or save the current value of all the selected device registers to a txt file by selecting the "Register dump " option from the menu. (Figure 19)

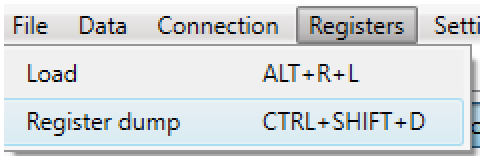


Figure 19: RoKiX Windows GUI Register menu [2].

The settings menu allows users to change multiple application settings on the user level. (Figure 20). The "Auto connect USB" menu item allows users to enable or disable to select the USB COM port for the connected device automatically. The "Auto config and registers download" menu item allows users to enable or disable the RoKiX Windows GUI download check for new configurations and registers from google cloud. "Automatic streaming" is used to enable or disable data streaming automatically. The "COM port" menu item is used to select the COM port when multiple devices are connected. "Reset connection" is used to reset and refresh the current connection.

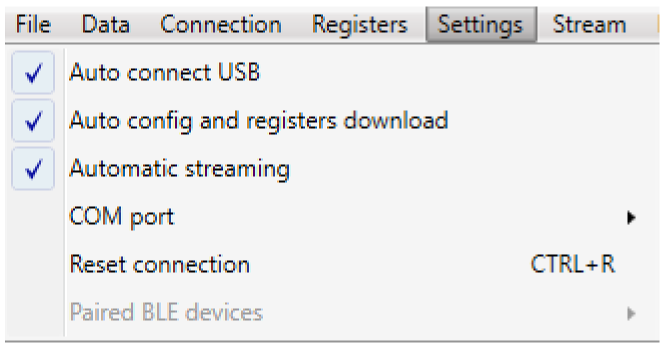


Figure 20: RoKiX Windows GUI Settings menu [2].

The view menu items provide different features for the RoKiX Windows GUI plotter and board configuration. (Figure 20) The "Sub channel view" and "Digital output in sub channel view" menu item is used to show or hide the sub channel view and digital output view for each subchannel. (Figure 21)

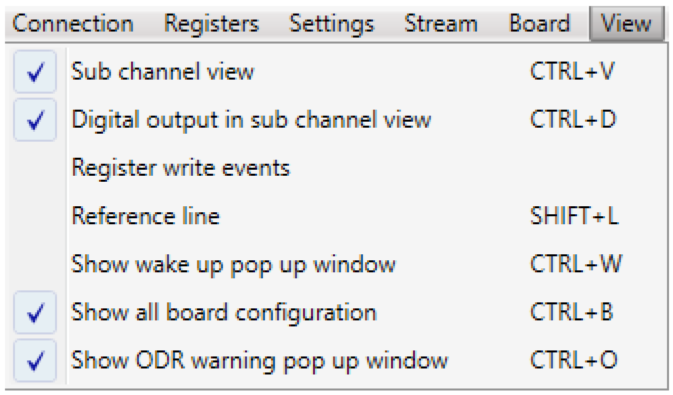
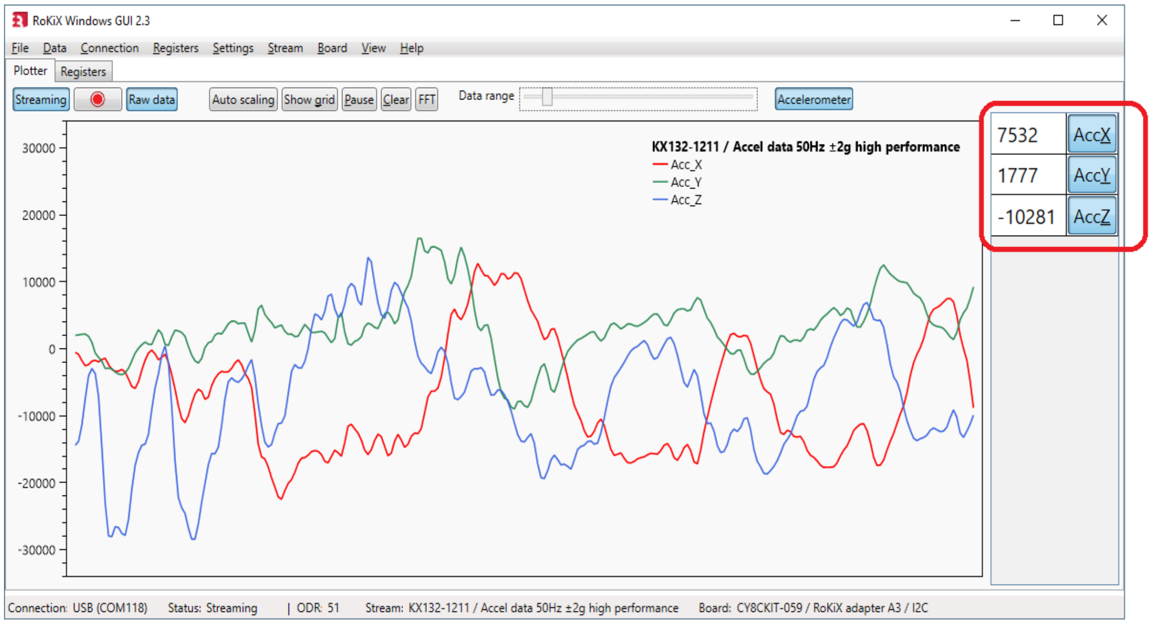

Connection
Registers
Settings
Stream
Board
View
✓ Sub channel view CTRL+V
✓ Digital output in sub channel view CTRL+D
Register write events
Reference line SHIFT+L
Show wake up pop up window CTRL+W
✓ Show all board configuration CTRL+B
✓ Show ODR warning pop up window CTRL+O

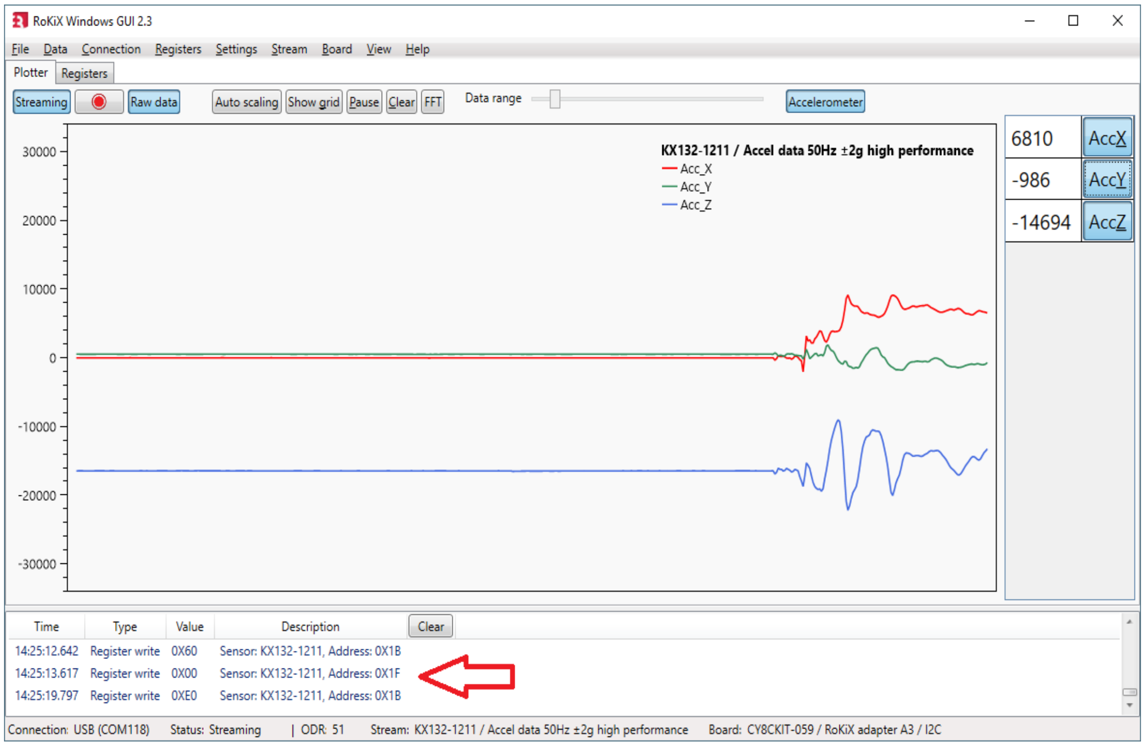
Figure 21: RoKiX Windows GUI view menu [2].


RoKiX Windows GUI 2.3
File Data Connection Registers Settings Stream Board View Help
Plotter Registers
Streaming Raw data Auto scaling Show grid Pause Clear FFT Data range Accelerometer
KX132-1211 / Accel data 50Hz ±2g high performance
Acc_X
Acc_Y
Acc_Z
7532 AccX
1777 AccY
-10281 AccZ

Connection: USB (COM118) Status: Streaming | ODR: 51 Stream: KX132-1211 / Accel data 50Hz ±2g high performance Board: CY8CKIT-059 / RoKiX adapter A3 / I2C

Figure 22: RoKiX Windows GUI sub-channel activated [2].

The “Register write events” allows users to show or hide the register write events window located below the plot window. (*Figure 23*)

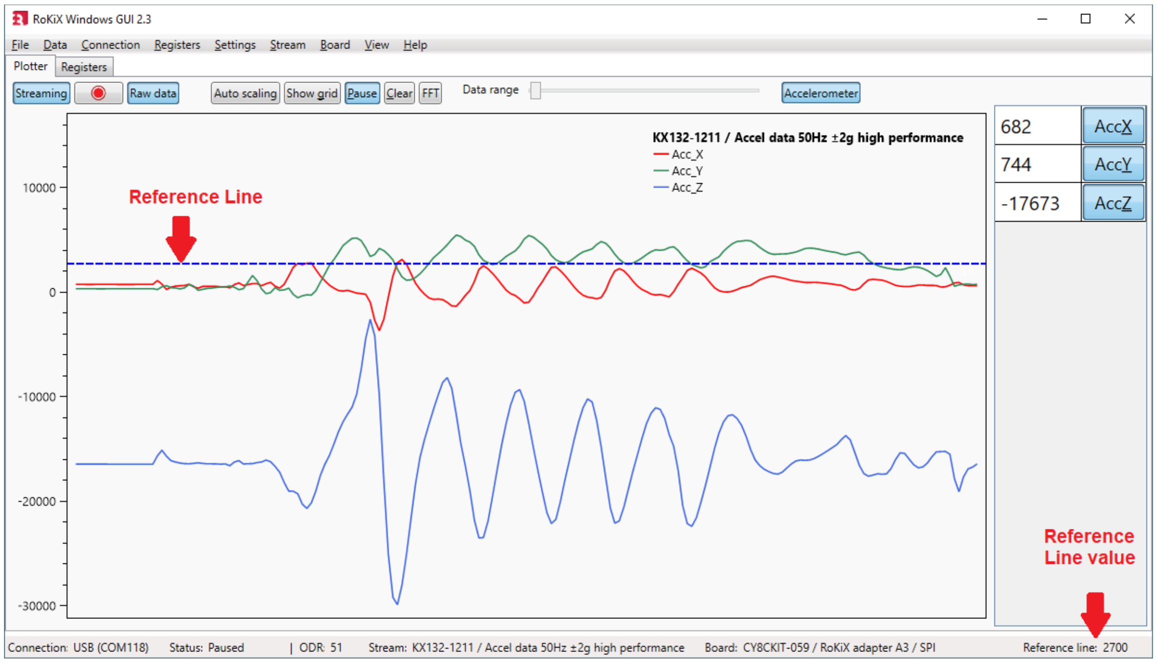

RoKiX Windows GUI 2.3
File Data Connection Registers Settings Stream Board View Help
Plotter Registers
Streaming Raw data Auto scaling Show grid Pause Clear FFT Data range Accelerometer
KX132-1211 / Accel data 50Hz ±2g high performance
Acc_X
Acc_Y
Acc_Z
6810 AccX
-986 AccY
-14694 AccZ
Clear

Time	Type	Value	Description
14:25:12.642	Register write	0X60	Sensor: KX132-1211, Address: 0X1B
14:25:13.617	Register write	0X00	Sensor: KX132-1211, Address: 0X1F
14:25:19.797	Register write	0XE0	Sensor: KX132-1211, Address: 0X1B

Connection: USB (COM118) Status: Streaming | ODR: 51 Stream: KX132-1211 / Accel data 50Hz ±2g high performance Board: CY8CKIT-059 / RoKiX adapter A3 / I2C

Figure 23: Register write Event output window [2].

The "reference line" menu item is used to show an additional horizontal line that can help to compare the real-time signal value against the reference value. The reference line value can be seen from the status bar at the bottom of the window (Figure 24).

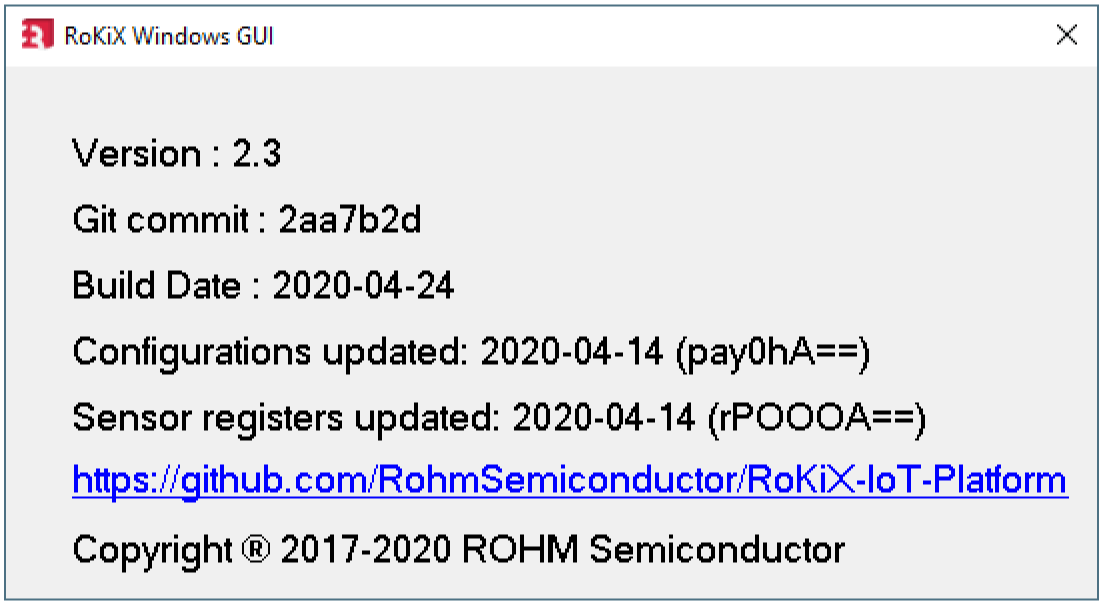

RoKiX Windows GUI 2.3
File Data Connection Registers Settings Stream Board View Help
Plotter Registers
Streaming Raw data Auto scaling Show grid Pause Clear FFT Data range Accelerometer
KX132-1211 / Accel data 50Hz ±2g high performance
Acc_X
Acc_Y
Acc_Z
Reference Line
10000
0
-10000
-20000
-30000
682 AccX
744 AccY
-17673 AccZ
Reference Line value

Connection: USB (COM118) Status: Paused | ODR: 51 Stream: KX132-1211 / Accel data 50Hz ±2g high performance Board: CY8CKIT-059 / RoKiX adapter A3 / SPI Reference line: 2700

Figure 24: Plotter view [2].

The "Show wake up pop up window" sub-menu item is only visible when selecting a wake-up / back-to sleep detection stream. A pop window will show up when the wake-up event is detected from the connected device when selected. The "Show all board configuration" allows users to show all supported board configurations for all supported host adapters or only the relevant ones supported by the host adapter currently connected. The "show ODR warning" allows users to show or hide warning pop up messages when the real-time Output Data Rate (ODR) as measured by the RoKiX Windows GUI is significantly different from the nominal ODR set in the stream.

The "About RoKiX Windows GUI" menu item shows detailed information about the current RoKiX Windows GUI version. (Figure 25)


RoKiX Windows GUI

Version : 2.3

Git commit : 2aa7b2d

Build Date : 2020-04-24

Configurations updated: 2020-04-14 (pay0hA==)

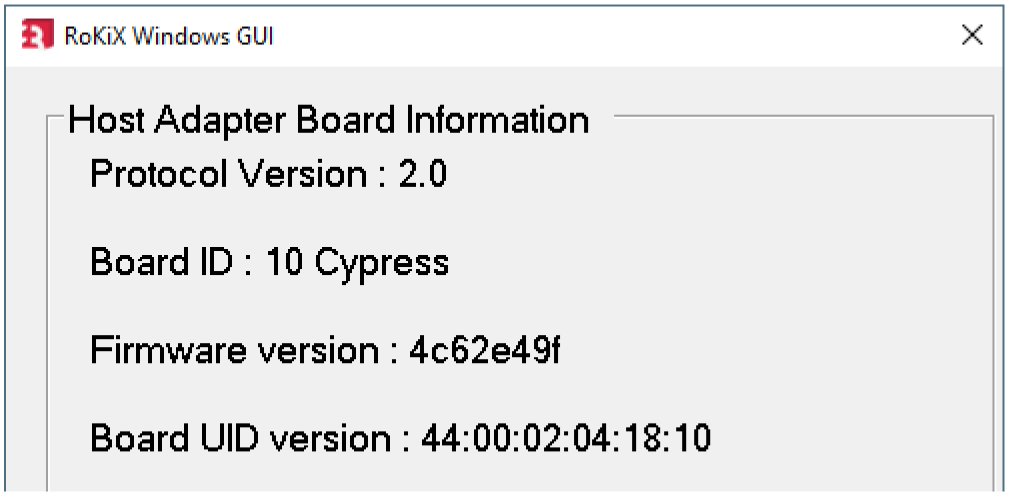
Sensor registers updated: 2020-04-14 (rPOOOA==)

https://github.com/RohmSemiconductor/RoKiX-IoT-Platform

Copyright ® 2017-2020 ROHM Semiconductor

Figure 25: About RoKiX Windows GUI about menu [2].

The “About Host Adapter Board” menu provides information about the connected device. (Figure 26)


RoKiX Windows GUI

Host Adapter Board Information

Protocol Version : 2.0

Board ID : 10 Cypress

Firmware version : 4c62e49f

Board UID version : 44:00:02:04:18:10

Figure 26: About Host Adapter menu [2].

4. Implementation (WinAppDriver)

This chapter discusses the implementation of a UI automated testing system for RoKiX Windows GUI. Section 4.1 presents the current state of testing of the RoKiX Windows GUI. It compares different structure-based tools that came into consideration when starting this project. Section 4.2 discusses the objective of the UI automated system. Also, we evaluate different ID-based testing tools. The UI automated testing system is assessed at the end of the chapter.

4.1 Current state of testing of the RoKiX Windows GUI

I am working on the RoKiX project for ROHM co. RoKiX Windows GUI is part of the RoKiX IoT platform Client software, which provides an easy-to-use graphical user interface demonstrating high-level device offering and features [2][3]. Previously, RoKiX Windows GUI has only been tested manually for each public release. Major public releases occur four times per year. Minor internal releases occur more frequently.

The testing process requires three to four engineers to test the software manually. Test case documents are prepared beforehand to determine different use cases for the tester to follow during testing. The test document is an excel file that includes the test case, and the test results are presented as fail or pass [2]. It contains all the test steps and test scenarios that the tester needs to follow. The testing process will take 4 to 6 hours for each engineer. In total, for a major release, the testing process will take 12 to 24 working hours. However, for a minor release, only the developer tests the software due to time constraints. As a result, minor releases might contain some bugs.

The first issue in the current testing is that it is time-consuming and expensive. The testing process takes the tester's time from other essential tasks. Since the current testing is costly, there has not been any regression testing. Thus, all the existing features have not been tested before each public release. The other major issue in the current testing system is quality. The quality of manual testing depends on the tester's motivation for finding defects in the software. Testing a software application multiple times affects the tester. This can lead to team demotivation, affecting the quality of the test result; reporting the test result is also an issue. Each tester uses the test case excel document to report their test result. The results are then sent to the developer. This process is ineffective and time-consuming.

RoKiX windows GUI is expanding and supporting more devices. As a result, minor releases have been made more frequently. Since minor releases are not tested, they might contain some defects. With the growing number of supported devices, the testing time has increased. The RoKiX teams decided that manual testing RoKiX Windows GUI is not practical for future releases.

4.2 Development of the test automation system

4.2.1 Evaluation of different UI automated testing methods and tools

From a technological point of view, structure-based testing is a better alternative than visual GUI testing. Structure-based testing has lower maintenance costs and is more reliable. However, the software testing structure may not always provide adequate access to some parts, and in those cases, visual GUI testing may be the only alternative. One requirement of this project is that the tool evaluated should be free to use.

Other methods and tools were evaluated during the early stage of this project. Capture & Replay based Test Studio, structure-based FlaUI, and WinAppDriver were the most extensive tools being assessed. Test Studio comes with a licensing fee. Using a trial version, Test Studio recorded one session of RoKiX Windows GUI. While recording, RoKiX Windows GUI was not responsive multiple times. We had to rerun the recording for a considerable time to finish one full session. This is due to RoKiX Windows GUI having a heavy Tree element structure. Hence, we decided not to use Test Studio. Table 2 shows the advantages and disadvantages of FlaUI and WinAppDriver.

Table 2. Comparison of FlaUI and WinAppDriver.

Tool	**Advantages**	**Disadvantages**
FlaUI	A free and open-source tool	It is not hosted on GitHub.

	Support for Windows 7,8 and 10	Scarce documentation and online support
WinAppDriver	A free and partially open-source tool Hosted on GitHub. Developed and supported by Microsoft. Documentation is easily accessible. Capable of controlling multiple machines using driver instances. Support touch devices Actively developed.	Only support Windows 10

The lack of documentation and support was the reason to choose WinAppDriver over FlaUI. One of the requirements of the project was that the implementation should be done in a short period. As such, online support was an essential factor.

4.2.2 WinAppDriver

Microsoft's WinAppDriver (Windows Application Driver) is a service to support Selenium-like UI Test Automation on Windows Applications [33]. It supports UWP (Universal Windows Platform), WinForms (Windows Forms), Classic Windows (Win32), and WPF (Windows Presentation Foundation). It compiles to the JSON Wire Protocol standard. Since WinAppDriver is derived from Selenium WebDriver, it supports multiple programming languages such as C#, Java, JavaScript, Python, and Ruby. It can also support numerous runners such as Junit that should be compatible with most CI setup or build systems. WinAppDriver comes with a UI recorder to inspect UI elements.

4.2.3 Objective of the UI automated system

The high-level objective of the user interface tests is to automate as much regression testing as possible. There are so many features that it is not feasible to manually test all of them, even in a simple way for each software version. The goal of this project is to

minimize manual testing. That way, it would be possible to redirect the manual testing to exploratory testing. The other objective is to have high test coverage with high-quality tests. The aim of the automated tests is defined and presented in Table 3.
The test reporting objective is to make the test result informative and make them useful for the developers. The test reporting goals are presented in Table 4.

Table 3. Objective of the UI automated testing.

Objectives	**Descriptions**
Ability to find defects	The ability to find defects is tangible evidence that the tests are useful and can prevent defects from being deployed in the production system. However, tests are valuable even if they do not find defects by providing some confidence in the software's correct functionality.
Implementation cost	The overall cost of UI automation is higher than manual testing now. However, in the long term, the overall cost of testing RoKiX Windows GUI must be lower than manual testing.
Test maintenance	The test case design must accommodate changes in the source code of the software under test. When the software under test source code is changed, the test's change should not require much of work.
Test extendibility	The tests should be extendible so that the developer can reuse some test scripts for other test cases. As such, it is easier to achieve good test coverage.
Test reliability	The test results should be consistent with the expected result. The test should fail if there is any defect detected.
Text execution time	The test execution time should not take too much time so that the feedback loop is shorter. There should also be an option to run the entire test case or individual test cases.

Table 4. Objective of the UI automated testing reporting.

Objectives	**Descriptions**
Test report quality	The test report must be clear and understandable. The report should state which test case failed or succeeded. The reason for the failures should also be included in the report.
Feedback loop	There should be a short feedback loop. When developers execute test cases, the report should automatically be updated and reported to the developers.
Test report accessibility	The test report should be accessible to all developers at once. It should be hosted on some cloud platform.

4.2.4 Overview of the UI automated testing system

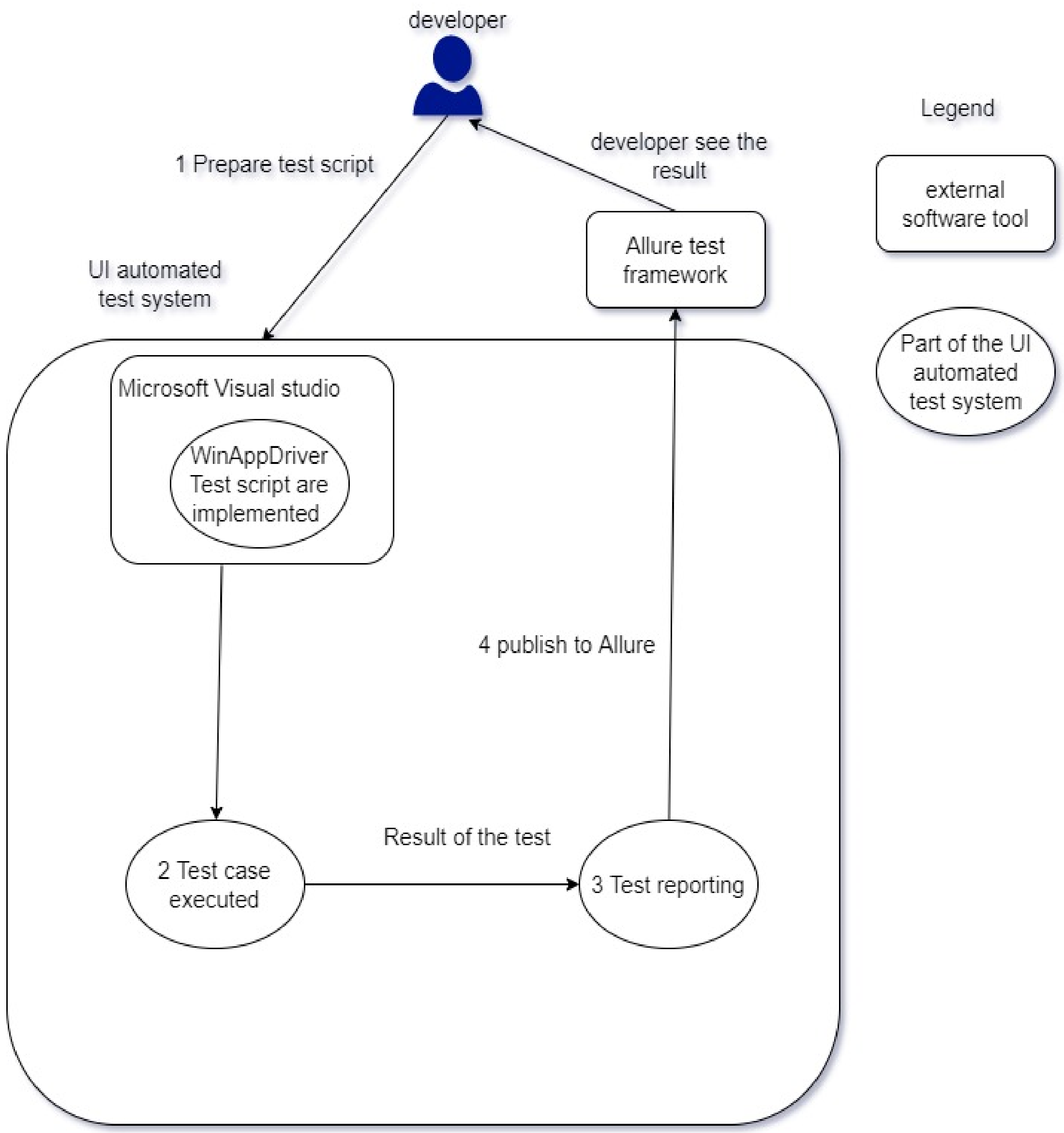


Figure 27: Architecture of the test automation system.

Figure 27 represents the architecture and current implementation of the UI automated testing system. Each stage of testing is described as follow:

1 Prepare A Test script:

A developer prepares a test script from the test cases. There are two big test cases. One test case covers all the UI elements in the plotter area, and the latter covers all the UI elements in the Register tab area. The two big test cases are divided into four smaller test cases. In total, there are 8 test cases. This will give the developer option to run a specific test case or run longer test cases. It also facilitates test case maintenance. [22] Figure 30 and Figure 31 shows the test coverage of the test cases. Based on Al´egroth et al. [6], we designed the test cases based on manual test cases. [6] Those test scripts are prepared using Microsoft Visual Studio 2017 and WinAppDriver. Before writing the test script, all UI elements in RoKiX Windows GUI are given an ID. Those ids are then verified by Inspect.exe, which is Microsoft's inspector tool. Each UI element is accessed using the FindElementByAccessibilityId method. Figure 28 shows an example using the FindElementByAccessibilityId method. [38] FindElementByAccessibilityId method checks if the UI element with ID "menu" exist in the session. The position of the element is not relevant.[33] TestFileExit() method is a simple test method that tests the functionality of the exit menu item. (Figure 16) When running this method, RoKiX Windows GUI starts running, and from the File menu, it clicks the exit menu item. Then the methods check if the exit menu item closes the application.

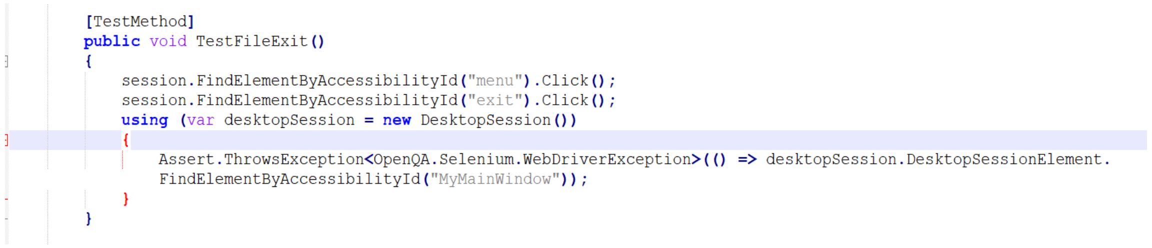

```
[TestMethod]
public void TestFileExit()
{
    session.FindElementByAccessibilityId("menu").Click();
    session.FindElementByAccessibilityId("exit").Click();
    using (var desktopSession = new DesktopSession())
    {
        Assert.ThrowsException<OpenQA.Selenium.WebDriverException>(() => desktopSession.DesktopSessionElement.
        FindElementByAccessibilityId("MyMainWindow"));
    }
}
```

Figure 28: An example test script using WinAppDriver. [33]

2 Test Case executed

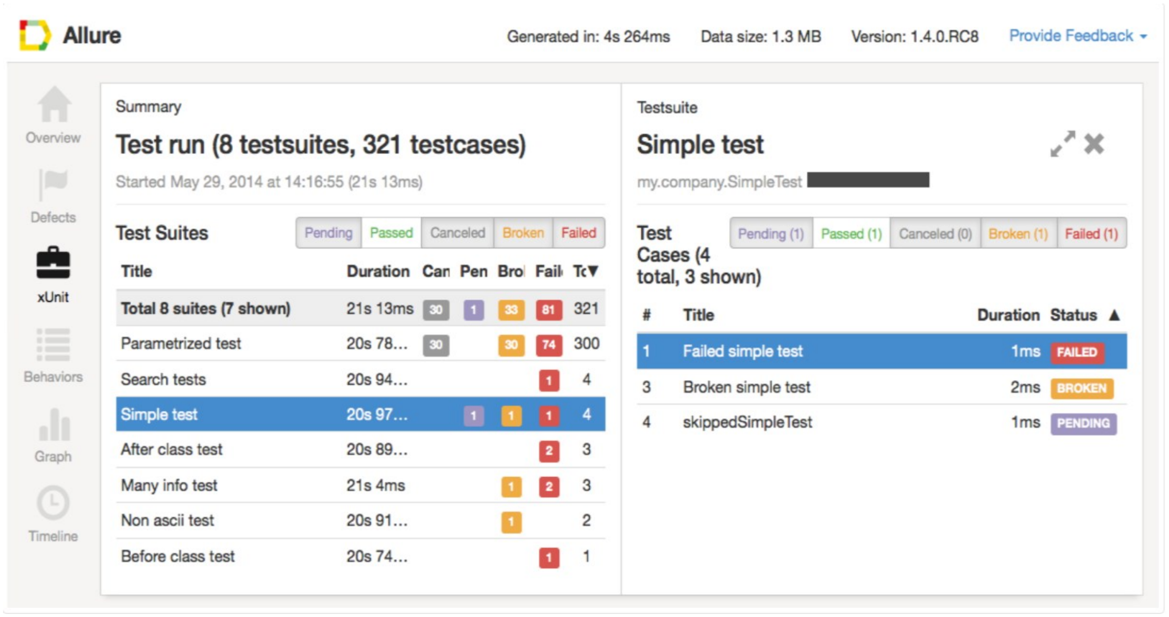
Batch files are then created for each test case. The test cases are run by executing the corresponding batch file. With the current implementation, the test cases are executed manually by the developer. This means that the developer needs to run the corresponding batch file to run a test case. Currently, the developer can execute all test cases using only one batch file. This will take less than an hour to complete. It is possible to run some subset of test cases. Failed test cases will not stop the testing process.

3 Test Reporting

After executing each test case, WinAppDriver generates the test results. Those test results are in trx format. A script is prepared to automatically create HTML pages with the date and time stamp as the file name.

4 Publish to Allure

The Allure test suite is an open-source framework that hosts test reports. The Allure test suite's test reports show a clear representation of what has been tested in a user-friendly form. [34] The test results are loaded to the Allure test framework using a script. The script utilizes the allure report generator tools to upload to the Allure cloud server.
It also allows everyone to see the result of the executed tests since it is easily accessible. [34] This script's purpose is to upload the generated test results into a repository in bitbucket. This repository is dynamically connected with the Allure test suite website.



Generated in: 4s 264ms Data size: 1.3 MB Version: 1.4.0.RC8

Summary

Test run (8 testsuites, 321 testcases)

Started May 29, 2014 at 14:16:55 (21s 13ms)

Test Suites

Title	Duration	Can	Pen	Bro	Fail	Tc▼
Total 8 suites (7 shown)	21s 13ms	30	1	33	81	321
Parametrized test	20s 78...	30		30	74	300
Search tests	20s 94...				1	4
Simple test	20s 97...		1	1	1	4
After class test	20s 89...				2	3
Many info test	21s 4ms			1	2	3
Non ascii test	20s 91...			1		2
Before class test	20s 74...				1	1

Testsuite

Simple test

my.company.SimpleTest

Test Cases (4 total, 3 shown) Pending (1) Passed (1) Canceled (0) Broken (1) Failed (1)

#	Title	Duration	Status ▲
1	Failed simple test	1ms	FAILED
3	Broken simple test	2ms	BROKEN
4	skippedSimpleTest	1ms	PENDING

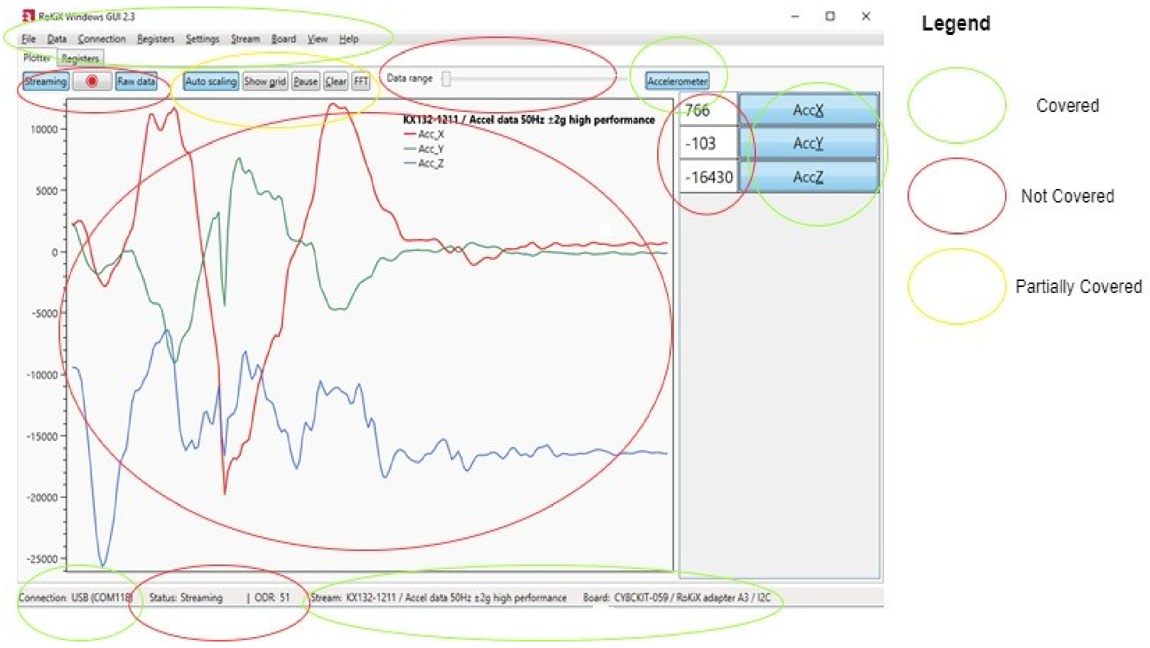
Figure 29 : Example of a test report using the Allure test suite [34].

4.2.5 Test coverage

The UI automated testing systems will only test the RoKiX Windows GUI in a disconnected state. The current test cases cover both Registers and Plotter tab. The elements test can be seen in Figure 30 and Figure 31. The functionality of each UI element is described in Chapter 3.

1 Plotter tab

Since the software under test is tested in a disconnected state, any dynamic values are not tested. The plotter and digital output data text fields are not tested. Also, the status of the connection is not tested, as shown in Figure 30. The plotter settings such as “Auto Scaling”, “Show grid”,” Pause”, “Clear”, and “FFT” button are tested partially. This means that the functionality of the buttons is not tested, but the UI response is tested. For example, the “Show grid” button is tested without the plotter.



Connection: USB (COM118) Status: Streaming | ODR: 51 Stream: KX132-1211 / Accel data 50Hz ±2g high performance Board: CY8CKIT-059 / RoKiX adapter A3 / I2C

Figure 30: Plotter tab Test coverage.

2 Register tab

Like the plotter tab, the register tab is tested in a disconnected state, as seen in Figure 31. As such, register Write, Read, and polling operations are not tested. The status, Value, and Bin Value of the register are partially tested. Only the POR and EDITED status are tested.

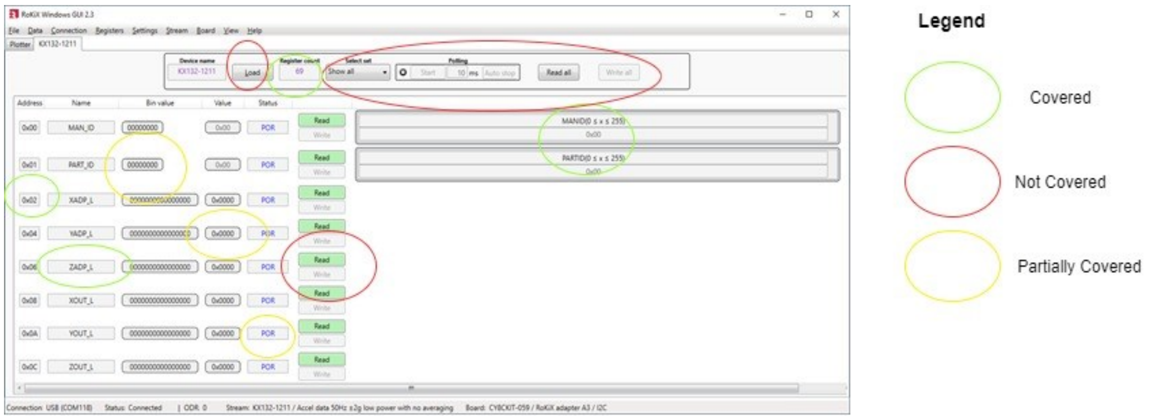


Figure 31: Register tab Test coverage.

4.3 Evaluation of the UI automated testing system

4.3.1 Tests

The tests were evaluated based on different criteria. The criteria were cost-effectiveness, reliability, extendibility, and execution time. They were described in detail in section 4.2.3 Table 3.

The development cost of new test cases depends mainly on how the tests are developed. We now have eight small reusable methods of testing. This facilitates maintaining old test cases in the future. Also, it gives an incentive to improve the test cases for future development. It is critical to pay attention to the test structure continually. Focusing on regression testing helps lower the maintenance cost since existing RoKiX Windows GUI features do not change frequently.

The automated testing system has detected eight defects or issues. These defects have been consistent. The result of these tests has not changed after performing repetitive tests. Most test cases take a reasonable time to complete (less than five minutes). This is due to executing specific test cases or executing the entire session, which takes less than one hour. Also, this is ensured by not adding unnecessary delays between various steps of the tests.

4.3.2 Tests Reporting infrastructure

Test reporting is evaluated based on the objective in section 4.2.3 Table 4. With the current implementation, the developer can clearly understand the test results. The reason for failure, the time, and the code line are clearly stated in the test result.
The current implementation of the test result is not entirely automated. A developer must manually run the batch to publish the test result. A solution to this issue is discussed in the next chapter.

5. Result

This chapter analyzes the result of the implementation. It also compares it with the previous testing method. The comparison criteria are cost-effectiveness, test result reliability, and quality. The limitation of this study is also discussed at the end of this chapter.

5.1 Cost-effectiveness

Since the current test cases are in a disconnected state, it is not feasible to automate all manual test cases. It is essential to decide which manual test cases to automate. In the RoKiX project, we focused on testing existing features such as Menu item bars, the plotter settings, and the register tab. These are existing features that will not change in the future. The test cases are designed to cover as many features as possible, such as a test case covering all the UI elements in the plotter area. In one session, a user can test all the UI elements in the plotter area. This will make the test cases achieve higher test coverage. Also, it is a practical strategy to find defects. Alégroth and Feldt [12] and Al´egroth et al. [6] highlight the importance of automated regression testing.

Table 5. Test Automation implementation hours log.

Task	**Estimate Hours**	**Spent Hours**
Project management and technical support	160	121
Allure documentation and setup	80	72
Plotter UI test	126	127
Register test	122	132
Test result improvement	80	122
Total	568	574

Since this case study used a free tool, the only cost of this automation system is the development cost. Table 5 presents the implementation hours log. In total, it took 574 hours to implement the UI automated testing system. Manually testing the RoKiX Windows GUI software currently takes three to four engineers five to six working hours when doing public releases with the previous testing system. The engineers' tasks include executing test cases and preparing the test result document. This means it will take 18 to 24 working hours. When we compare the working hours of the manual testing working time with the automation testing, the difference is significant. The automation system development cost is 24 times more expensive than manual testing for the first deployment. When comparing the cost of these two methods, we should also consider the test's frequency, the code change of the software under test, and the test result quality.

As previously mentioned, the main feature of RoKiX Windows rarely changes. As such, the automated testing system maintenance hours and cost will be low in the future. Since the RoKiX Windows GUI is expanding and supporting more devices, the internal release will increase shortly. This will increase the test frequency. To further evaluate the automated test system effectiveness, we need to calculate the Return Of Investment (ROI). Figure 32 shows the equation to calculate the ROI.

$$\text{Automation ROI:} \quad \frac{\text{Gains} - \text{Investment}}{\text{Investment}}$$

Figure 32: ROI equation [35].

The investment and the gains can be calculated directly by comparing the manual testing cost and the automated testing cost. Since the hourly wages of both testing methods are the same, we can now compare the testing time. The automated testing method is 24 times more expensive than manual testing. This means that the company will save money after making 24 RoKiX Windows GUI releases. Figure 33 is a chart that shows the monetary ROI from investing in this case study automation. The company has planned to make a release every two months. This means over eight years; Rohm can save up to 600 hours. Figure 33 only shows the quantitative gains of using the automated testing system.

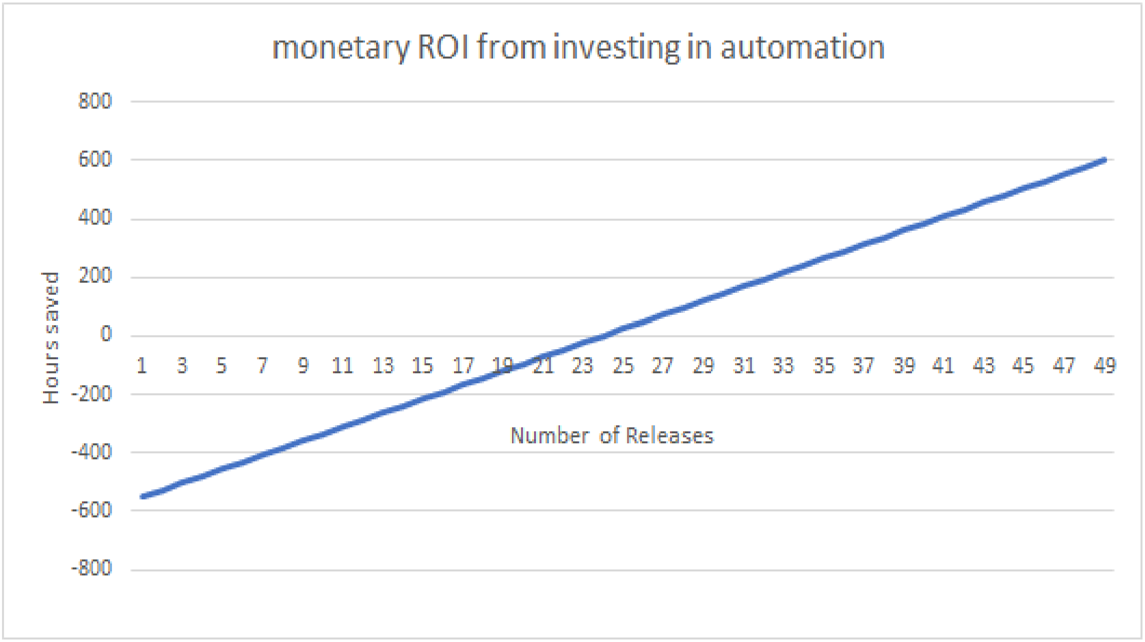


Figure 33: Monetary ROI from investing in automation.

Some of the qualitative benefits include reduction of defect leakage and a high-quality test reporting mechanism. “Defect leakage refers to the number of bugs or issues that end up in production due to not being found earlier in the software development lifecycle” [35]. The automated testing result reporting has improved compared to the manual test result reporting. The former method saves time and money. Another aspect that needs to be considered is the developers’ demotivation. Deak et al. [36] study results show that automating manual tests can increase developers’ satisfaction.

5.2 Limitation, challenge, and improvement

The literature review covers various test automation methods, and numerous findings are based on a few referenced studies. The literature review is mostly based on Alegroth et al. [32]. Other authors mentioned in this literature review are Maurizio Leotta, Arvid Karlsson, and Alexander Radway. While it would be good to have more referenced research, all of them are considered reliable authors, particularly Leotta and Alegroth, since they have both written their PhD thesis on automated software testing. Although RoKiX Windows GUI is a Windows application, most of the examples discussed in the literature review are web-based. While it would be good to have more Windows-based examples, web-based UI automated testing methods are similar to windows-based applications [8].

The evaluated tools are all free since it was one of the requirements of this case study. These requirements limit the number of tools to be used. Various commercial software testing tools can yield better results than WinAppDriver, notably TestComplete. The major advantage of using TestComplete over WinAppDriver is support for dynamic UI elements. TestComplete single license costs $4,600 and a concurrent license costs $9,000.

As mentioned in Chapter 4, the current test cases are in a disconnected state. The plotter and digital output data text fields are not tested. It is possible to use one of the supported devices to RoKiX Windows GUI to test the plotter and the register read and write operations. However, we wanted to make the test cases independent of the device connected. To solve this issue, one team member of the RoKiX team has developed a hardware emulator. This will enable the testing to be done in a closed system, not involving actual hardware. It is useful to use the hardware emulator when there is a limited number of specific devices to be tested, and the device output can be controllable during testing. It also maximizes test coverage. The hardware emulator will be used in the next release.

Previously, test results were made manually in excel format. The previous test result only contains issues and bugs described by the tester. Our current test result specifies the error message, failed tests, and the error stack trace, including the date and time. This information is helpful for the developer to fix the issue found promptly. The test results are also collected faster, since the results are published automatically when the test is executed. Detecting errors earlier is essential in maintaining test quality. This will result in having a shorter feedback loop. Our current automated test is executed using a batch file. It is not an automatic unmonitored test execution yet. If the issues detected are not fixed on time, then the technical debts can increase, making it more challenging to fix them later.

The automated testing system is not yet integrated into a Continuous Integration (CI)/ Continuous Delivery (CD) pipeline. With CI/CD pipeline, the overall execution time will be shorter. Figure 34 shows the azure pipeline flow. With the azure CI/CD, developers will only make a commit, then the automated test will run in parallel, and the result and other setups will be done automatically. Additionally, other tasks such as pre-deployment approval and deployment can also be done in CI/CD pipeline. There are, however, some challenges that come with CI/CD. To set up the CI/CD pipeline, more time is required, and the overall cost of the automation could be higher.

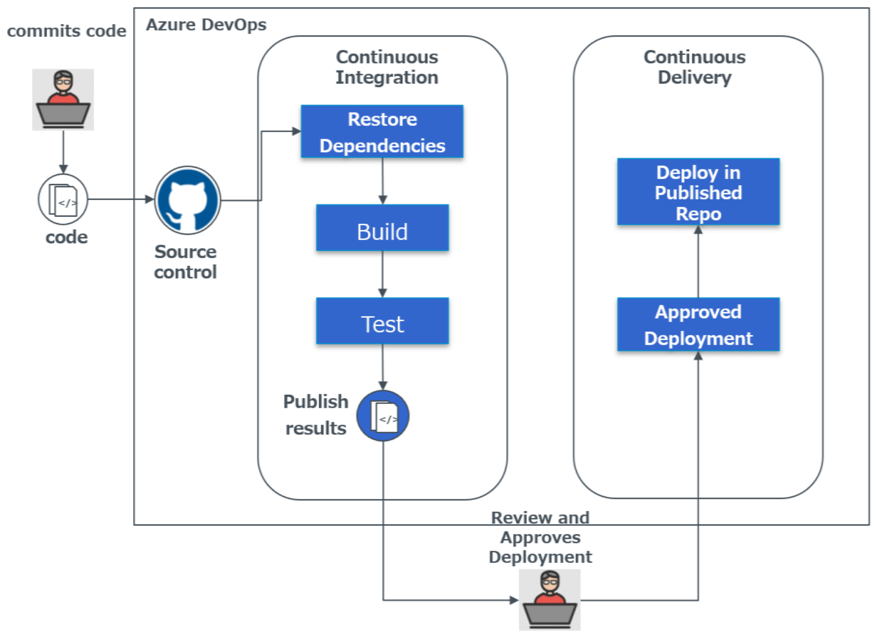


Figure 34: Azure pipeline flow.

6. Conclusion

This thesis evaluated different automated testing methods, including manual testing for RoKiX Windows GUI. The literature review compared in detail the software testing life cycle. It presented various studies about visual GUI testing, model-based testing, and structure-based testing. The literature review also evaluated the advantages and disadvantages of these different automated testing methods and various UI automated testing tools.

The literature review results indicate that structure-based testing is the recommended method for RoKiX Windows GUI's UI automated testing system. The case study evaluated different structure-based testing tools. Structure-based testing is reliable and less expensive than other testing methods. Based on various evaluation testing tools and the project requirements, WinAppDriver was chosen.

The practical part of the thesis began with an overview of RoKiX Windows GUI. The different parts of the RoKiX Windows GUI UI element are discussed in detail. Additionally, the project objective and requirements were discussed. The implementation starts by presenting the current RoKiX's testing method. It then gives an overview of the UI automated testing system. Each step of testing is then discussed further. The test coverage is also discussed in this chapter. The result of this case study is then presented in the last chapter.

The aim of this thesis is to introduce and evaluate a UI automated testing system for RoKiX Windows GUI. The result of the thesis result indicates that for the tests to be useful and save time and money, the testing should cover regression tests as much as possible. We observed that even simple test cases could help discover regression errors. Additionally, the test reports need to be clear and informative. This is important, since it will help in maintaining good test quality. It is also essential to have a good test architecture and design to allow the code to be reused. This will lower the testing maintenance cost.

The RoKiX team will try to improve this UI automated testing system for future releases. The team is considering replacing WinAppDriver with testComplete. There is also a plan to integrate the testing system into CI/CD pipeline. Since all these improvements increase the cost, the team needs to evaluate and plan carefully.

Based on this research, it is difficult to conclude that UI automated testing will save time and money for other ROHM software applications. The software under test needs to be evaluated before introducing a UI automated testing system. Based on this case study, the right software candidate for automation is released frequently and frequently does not require changing its existing features.

REFERENCES

[1] Grechanik, M., Xie, Q. and Fu, C. Experimental Assessment of Manual Versus Tool-Based Maintenance of GUI-Directed Test Scripts. *In IEEE International Conference on Software Maintenance.* IEEE. ISBN 1424448972 [2009] pp. 9-18.

[2] RoKiX Development Kit User Guide https://github.com/RohmSemiconductor/RoKiX-IoT-Platform/blob/master/RoKiX-Development-Kit-User-Guide.pdf. Accessed 20.09.2020

[3] Developers tools https://www.kionix.com/developer-tools. Accessed 20.09.2020

[4] Laurence R Kepple. The black art of GUI testing. *Dr. Dobb's Journal-Software Tools for the Professional Programmer*, 19(2):40–47, 1994.

[5] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. *In Proceedings of the 7th International Workshop on Automation of Software Test,* pages 36–42. IEEE Press, 2012

[6] Emil Al´egroth, Robert Feldt, and Lisa Ryrholm. Visual gui testing in practice: challenges, problems, and limitations. *Empirical Software Engineering*, 20(3):694–744, 2015.

[7] Kaner, C. A tutorial in exploratory testing. In QAI QUEST Conference [2008] pp. 36- 41. Visited: 2014-06-12.
URL: http://kaner.com/pdfs/QAIExploring.pdf

[8] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. *Visual vs. dom-based web locators: An empirical study. In International Conference on Web Engineering*, pages 322–340. Springer, 2014.

[9] Jovic, Milan & Adamoli, Andrea & Zaparanuks, Dmitrijs & Hauswirth, Matthias. (2010). *Automating performance testing of interactive Java applications*.

[10] Potter, R. Triggers: Guiding Automation with Pixels to Achieve Data Access. In A. Cypher, D.C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, BA. Myers and A. Turransky, eds., *Watch What I Do. Cambridge*, MA, USA: MIT Press. ISBN 0-262-03213-9 [1993] pp. 361{380.

[11] Jesper Lehtinen. *Automated gui testing of game development tools*, bachelor thesis at metropolia university of applied sciences. 2016.

[12] Alégroth, E., Feldt, R. On the long-term use of visual gui testing in industrial practice: a case study. *Empir Software Eng* 22, 2937–2971 (2017). https://doi.org/10.1007/s10664-016-9497-6

[13] Arvid Karlsson and Alexander Radway. Visual gui testing in continuous integration. Master's thesis, Chalmers University of Technology, University of Gothenburg, 2016

[14] Ina Schieferdecker. Model-based testing. IEEE software, 29(1):14, 2012.

[15] Uzun, B., Tekinerdogan, B., 2018. Model-driven architecture based testing: A systematic literature review. *Inf. Softw. Technol*. 102, 30–48.

[16] Utting, M., Pretschner, A., Legeard, B. A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 2011

[17] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[18] Neto, Arilo & Subramanyan, Rajesh & Vieira, Marlon & Travassos, Guilherme. (2007). *A survey on model-based testing approaches: a systematic review*. 31-36

[19] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," *2012 7th International Workshop on Automation of Software Test (AST)*, Zurich, 2012, pp. 36-42

[20] IS "MANUAL TESTING DYING"? AN INNOCENT QUOTE THAT HIT A NERVE
https://www.testcraft.io/manual-testing-dying-quote-hit-nerve/. Accessed 20.11.2020

[21] Manual testing process
https://medium.com/oceanize-geeks/manual-testing-process-340173d40141. Accessed 20.11.2020

[22] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing. John Wiley & Sons, Hoboken*, New Jersey, 2004.

[23] Jamil, Abid & Arif, Muhammad & Abubakar, Normi & Ahmad, Akhlaq. (2016). *Software Testing Techniques: A Literature Review*. 177-182.

[24] Juha Itkonen and Kristian Rautiainen. Exploratory testing: a multiple case study. In Empirical Software Engineering, 2005. International Symposium on Empirical Software Engineering, pages 10–pp. IEEE, 2005.

[25] James A Whittaker. What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79, 2000.

[26] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and lessons learned from automated testing. In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pages 571–579. IEEE, 2005.

[27] Arvid Karlsson and Alexander Radway. Visual gui testing in continuous integration. Master's thesis, Chalmers University of Technology, University of Gothenburg, 2016.

[28] CALP, M. Hanefi & Köse, Utku. (2019). Planning Activities in Software Testing Process: A Literature Review and Suggestions for Future Research.

[29] Pros and cons of "Record and playback" automated testing
https://www.deriskqa.com/Article-Pros-and-Cons-of-Record-and-Playback-Testing.html.

Accessed 24.11.2020

[30] what is structural testing in software testing?
https://www.testbytes.net/blog/structural-testing-in-software-testing/. Accessed 24.11.2020

[31] The challenges and benefits of Model-Based Testing
https://saucelabs.com/blog/the-challenges-and-benefits-of-model-based-testing. Accessed 24.11.2020

[32] Leivo, Teemu. "Automating user Interface testing: Case study at Finnish Transport Agency." (2017).

[33] Windows Application Driver
https://github.com/microsoft/WinAppDriver. Accessed 07.12.2020

[34] Allure Test Report
http://allure.qatools.ru/. Accessed 17.12.2020

[35] 6 Ways to Measure the ROI of Automated Testing
https://smartbear.com/resources/ebooks/6-ways-to-measure-the-roi-of-automated-testing/. Accessed 30.12.2020

[36] Deak, A., T. Stålhane and G. Sindre. "Challenges and strategies for motivating software testing personnel." *Inf. Softw. Technol.* 73 (2016): 1-15.