Aalto University

School of Electrical Engineering

Degree Programme in Automation and Systems Technology

Sakari A. Pesonen

An Open and General Numerical Control and Machine Vision Based Architecture for Payment Terminal Acceptance Test Automation

Master's Thesis

San Jose, Sept 27, 2016

Supervisor: D.Sc. Seppo Sierla, Aalto University

Advisor: M.Sc. Tatu Kairi

Aalto University
School of Electrical Engineering
Degree Programme in Automation and Systems Technology

ABSTRACT OF MASTER'S THESIS

Author:	Sakari A. Pesonen		
Title: An Open and General Numerical Control and Machine Vision Based Architecture for Payment Terminal Acceptance Test Automation			
Date:	Sept 27, 2016	**Pages:**	vii + 62
Major:	Intelligent Products	**Code:**	T-110
Supervisor:	D.Sc. Seppo Sierla		
Advisor:	M.Sc. Tatu Kairi		

Software testing is a crucial part of modern software development and it is commonly accepted fact that the earlier software defects and errors are found, the lower the cost of correcting those will be. Early detection of errors also increases the possibility to correct them properly.

Acceptance testing is a process of comparing the developed program to the initial requirements. Acceptance testing of a system should be executed in an environment as similar as possible to the production environment of the final product. This master's thesis will discuss how to address these in automated acceptance testing environment of payment terminal software.

This master's thesis will discuss the theories related to software testing, testing of embedded systems and the challenges related to the topic. Master's thesis will present a proposed architecture for automated acceptance testing of payment terminals including the needed hardware and software.

Keywords:	Automated Acceptance Testing, Software Testing, Payment Terminal, Robot Framework, Computer Vision, Open Source
Language:	English

Aalto-yliopisto
Sähtkötekniikan korkeakoulu
Automaatio- ja systeemitekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Sakari A. Pesonen		
Työn nimi: Avoin ja yleispätevä numeeriseen ohjaukseen ja konenäköteknologioihin pohjautuva maksupäätteiden automaattisen hyväksymistestausympäristön arkkitehtuuri			
Päiväys:	27. syyskuuta 2016	**Sivumäärä:**	vii + 62
Pääaine:	Älykkäät tuotteet	**Koodi:**	T-110
Valvoja:	TkT Seppo Sierla		
Ohjaaja:	FM Tatu Kairi		

Ohjelmistotestaus on tärkeä osa modernia ohjelmistotuotantoa ja on yleisesti tunnustettu, että mitä aiemmin virheet ohjelmistosta löytyvät, sitä edullisempaa niiden korjaaminen tulee olemaan. Aikainen virheiden havaitseminen myös edesauttaa virheiden perusteellista ja laadukasta korjaamista.

Hyväksymistestaus on ohjelmistotestauksen vaihe, jossa kehitettyä ohjelmistoa verrataan alkuperäisiin ohjelmistovaatimuksiin. Ohjelmiston hyväksymistestaus tulisi suorittaa lopullista tuotantoympäristöä mahdollisimman hyvin vastaavassa ympäristössä. Tämä diplomityö käsittelee näitä ohjeistuksia maksupäätteiden automaattisen hyväksymistestauksen ympäristössä.

Tämä diplomityö käsittelee ohjelmistotestaukseen liittyvää teoriaa, sulautettujen järjestelmien testausta sekä aiheeseen liittyviä haasteita. Lisäksi diplomityö esittelee ympäristön maksupäätteiden automaattiseen hyväksymistestaukseen ja käsittelee siihen tarvittuja ohjelmistoja ja fyysisiä komponentteja.

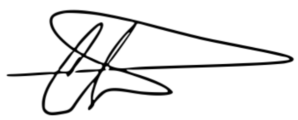
Asiasanat:	Automaattinen hyväksymistestaus, ohjelmistotestaus, maksupääte, robot framework, konenäkö, avoin lähdekoodi
Kieli:	Englanti

Acknowledgments

I wish to thank my instructor Tatu Kairi and my supervisor Seppo Sierla for their great help and knowledge throughout the writing process of the master's thesis.

I would also like to thank my manager Marko Klemetti for encouraging the writing process of this master's thesis. My dearest thanks go towards my family and the members of !nerdclub for their great support throughout my studies and the process of writing this master's thesis.

San Jose, Sept 27, 2016



Sakari A. Pesonen

Abbreviations and Acronyms

AAT	Automated Acceptance Test
BDD	Behavior-Driven Development
BW	Black and White
CCR	Cartesian Coordinate Robot
CNC	Computer Numeric Control
HMI	Human Machine Interface
LCD	Liquid Crystal Display
MDF	Medium-Density Fibreboard
NFC	Near Field Communication
OCR	Optical Character Recognition
PIN	Personal Identification Number
PLA	Polylactic Acid
PWM	Pulse Width Modulation
QA	Quality Assurance
RF	Robot Framework
SUT	System Under Test
UI	User Interface

Contents

Chapter 1

Introduction

Software testing is a crucial part of modern software development and it is a commonly accepted fact that the earlier defects and errors in the software are found, the lower the cost of correcting those will be. Early detection of errors also increases the possibility to correct them properly. (Myers et al., 2011)

Acceptance testing is the process of comparing the developed program to the initial requirements of the software (Myers et al., 2011). Automated acceptance testing (AAT) process should be executed whenever new features are added. Therefore, especially in agile software development, AAT plays an important role as new versions of software are being developed rapidly. Automation can free valuable human resources from this process (Haugset and Hanssen, 2008) and therefore lower the overall cost of the software.

According to Sommerville (2011), acceptance testing of a system should be executed in an environment as similar as possible to the production environment of the final product. System should also be tested with real data rather than with a simulated sample. When software is developed for an embedded system and therefore the production environment is an actual device, also the acceptance testing should be executed on genuine device with actually interacting through the user interface (UI) of the machine. Especially when testing embedded software, this leads to a situation where aspects pointed out above are in fact being emphasized, as late detection of defects in embedded software can considerably raise the overall cost of the system (Ebert and Jones, 2009).

Sommerville (2011) states that it is practically impossible to perfectly replicate the system's working environment. When considering an embedded system, this can be even harder. Buttons of the device have to be actually pressed and visual changes on the screen of the device have to be observed. In order to automate this, a testing environment has to be implemented that can observe and manipulate the device through the real physical user interface, i.e. not simulating the keystrokes nor reading the LCD communication line. The testing environment has to incorporate both hardware and software solutions to mimic real human user as realistically as possible.

This master's thesis will discuss the theories related to software testing, testing of embedded systems and the challenges stated above. In addition, this master's thesis presents an architecture for automated acceptance testing of payment terminal software including the needed hardware and software.

Research presented in this master's thesis was carried in co-operation with Eficode Oy and Nets Oy. Internationally Nets is one of the main payment terminal software providers in the Nordic countries.

1.1 Problem Statements

In order to survey the topic of this work at an adequate level, this master's thesis presents five different problem statements. Problem statements are as follows:

1. What are the benefits of using open source software and how can the architecture be designed to maximally exploit these benefits?
2. What are the distinguishing characteristics between different payment terminals that have impact on automated acceptance testing and how can the architecture be designed to adapt these with minimal effort?
3. What kinds of test automation approaches exist and which approach is best suited for payment terminal acceptance test automation?
4. How should the test syntax be defined in order to make the test suites compact and understandable while accommodating the needs of different payment terminals?

1.2 Structure of the Master's Thesis

This master's thesis first discusses the theories and literature related to the topic and then presents an architecture of automated test environment for payment terminal software acceptance testing. In the first Chapter of this master's thesis, the topic is introduced, problem statements are presented and structure of this work is explained.

Second Chapter covers the literature review of the topic of the master's thesis. Each problem statements have related sections and individual problem statements are being discussed in those sections. Each section first gives an introduction from problem statement's point of view followed by the most relevant references around the topic. Sections analyze what has been done earlier and how the fundamental aspects of these previous works can be used as a basis for this work.

Third Chapter of the master's thesis presents the proposed architecture for automated acceptance test environment for payment terminal software based on the literature review done in the previous Chapter. Chapter presents the fundamental parts of hardware and software needed for this kind of an environment. This section has diagrams of proposed software architecture as well as fundamental design of the needed hardware.

Fourth Chapter describes what was needed in order to achieve the AAT environment described in the previous Chapter. Different hardware and software subsystems of the AAT environment are presented and described. Implemented AAT environment and its subsystems are visualized in this Chapter using images and diagrams.

Fifth Chapter discusses the presented problem statements based on the proposal and implementation of the AAT environment described in the previous Chapters. Future research topics are also presented related to each problem statement.

Sixth and the final section concludes the research done on this master's thesis and will summarize the benefits obtained by this kind of an environment.

Chapter 2

Payment Terminal Acceptance Testing

When developing software with agile methodologies for payment terminals, i.e for an embedded system, testing is a crucial part of the process. The earlier the defects and errors in the software are detected, the lower the cost and needed effort will be for correcting those (Myers et al., 2011).

Test environment that can be used in acceptance testing of payment terminals has several challenges to tackle and matters related to physical and technical aspects of the payment terminals have to be considered. This Chapter discusses the background of these challenges. Open source technologies were also preferred by the customer. Therefore, a section discusses the benefits obtained by using open source software and hardware in acceptance testing environment for payment terminals. Chapter also discusses the different approaches for acceptance testing as well as how should the test suites be defined in order to make them understandable and reusable.

2.1 Benefits of Open Source Solutions

When designing an automated acceptance testing environment from scratch, evaluation and availability of different possible components play a significant role in terms of development speed and cost. Suitability of one individual software subsystem is hard to determine just based on a manual or documentation of the product. Software has to be evaluated in terms of functionality, stability and performance. In addition, different software decisions have to be compatible with each other. Software components might also need some modification to suit the needs of the intended environment. All this applies to the hardware parts as well.

Open source software provides an advantage on these matters over closed source products as the source code is easily available (Morgan and Finnegan, 2007). As open source software can be accessed free of charge, a component can be easily evaluated by trying out whether they work for the purpose or not. The evaluation can also include an analysis about how easily the open source product can be modified to suit the needs. This especially is hard to achieve with commercial closed source products as the source code is not available.

According to Paulson et al. (2004), open source projects usually have fewer defects than closed source projects. Defects are found and fixed rapidly as they are reported openly to the open source community. If a defect is found during evaluation of the product, it can also be corrected by the user. By doing this, the user can contribute to the project. This, on the other hand, is hardly never possible with closed software. Paulson et al. (2004) also state that open source projects foster more creativity than closed source counterparts. This means that number of functions added over time is higher in open source projects. When using the product in some new field of use, this can be a great advantage as user can report desired features to the community and it can be added relatively quickly if the feature is considered needed by the community.

Open-source hardware means that details and plans of the product and its parts are commonly available (Rubow, 2008; Acosta et al., 2009). This allows that parts can be manufactured and modified by anyone with knowledge and skills to suit individual needs. When detailed part descriptions are available, multiple manufacturers can fabricate the actual parts. This creates competition and therefore usually lowers the price of individual hardware components.

As the overall security of the payment terminals is a high priority, use of open source technologies is seen as an effort to fulfill this requirement. Open source products provide transparency to the actual users and therefore support growing trust amongst customers.

2.2 Common Characteristics Between Payment Terminals

When designing automated test environment for different kinds of payment terminals, different physical and technical features have to be taken into account. Environment has to be able to manipulate different types of payment terminals and test structure has to be designed to adapt to the needs of different software and their different versions running on the payment terminals.

Majority of payment terminals share some common characteristics as they are made for same purpose: handling card payments. Scope of this thesis is to propose a testing environment for those payment terminals that share three main features: a keyboard, a screen and a card slot. Different types of terminals are visualized in Figure 2.1 and Figure 2.2 below.

Screens of the payment terminals differ in terms of size, placement and type. Test environment has to take into account different screen placements and it has to support both black and white (BW) and colored displays.

Keyboards of payment terminals share majority of keys together as number keys are needed for entering the PIN code and accept- and decline-buttons are needed for accepting and canceling the payment. Keyboard layouts, however, differ between different manufacturers and even amongst different models of the same manufacturer.

Location of the chip card slot is usually on the lower side of the payment terminal or on top of the screen of the payment terminal. Research done within this master's thesis is limited to those terminals that have the chip card slot at the lower side of the payment terminal as this simplifies the hardware needed for test environment. This is described more in depth in section 3.2. This study is also limited to chip card readers and therefore, magnetic stripe readers and near field communication (NFC) payments are not addressed.

(a) Terminal 1

(b) Terminal 2

Figure 2.1: Two examples of payment terminals from different manufacturers. Image for subfigure 2.1a: (Ingenico payment terminal, n.d.)

Figure 2.2: Example of a payment terminal which attaches to a smart phone.

2.3 Different Approaches for Test Automation

According to Broekman and Notenboom (2003), testing of embedded systems and embedded system software can be very different depending on what kind of system is under testing. Mobile phones have to be tested in a very different manner than for example cruise control system in cars. Nevertheless, some general guidelines and similarities exists and should be followed.

Testing of a payment terminal software in an automated way can be viewed at different levels. Most abstract classification can be seen if the testing is divided into two levels: white box testing and black box testing. White box testing is a methodology where the source code is investigated and test cases are written to test the internal logic of the program. Black box testing, on the other hand, concentrates only on the inputs and the outputs of the software. Everything between those is not in the field of interest as black box testing only focuses on whether the right input produces the wanted output. (Nguyen, 2001; Myers et al., 2011)

Khan and Khan (2012) distinguishes these methodologies clearly from each other by stating that white box testing is a process wher,e full knowledge of source code is needed in order to write the tests. Black box testing is described in a way that only the inputs and outputs of the application has to be known and black box testing has no or only little relevance to internal works of the program (Pressman, 2005). Black box testing methodologies can be thus seen to apply for testing of working product against the initial requirements of the software.

Huizinga and Kolawa (2007), on the other hand, presents that test automation can be divided into several layers that are unit testing, integration testing, system testing and acceptance testing. Unit testing is defined to cover testing of a single unit of the software's source code e.g. individual methods and functions of the software. Integration testing is described as a testing phase to verify that different parts of the software work together as a group. System testing is described as being a testing phase where hardware and software is integrated and tested to meet the requirements of the system. This can however include simulated data. Acceptance testing is represented as highest abstraction level of this classification as it ensures that the final product meets its acceptance criteria defined by the customers.

However, these classifications are not mutually exclusive as both white box and black box testing methodologies can be applied to all levels of testing. For example, when implementing unit tests for a software, individual methods are commonly being tested in terms of whether a certain input produces a right output. This can be seen to follow the black box testing methodology if the methods tested are simple and small enough. Correspondingly, acceptance tests can be used to validate whether the system meets the business requirements and for this, knowledge about the business logic is needed (Haugset and Hanssen, 2008). This, on the other hand, can be seen to follow the white box testing methodology.

As black box testing is based on the external exceptions and behavior of the software (Khan and Khan, 2012), required acceptance testing of the payment terminal software can be seen to follow this methodology. Intended automated testing of the payment terminals seems to also follow the acceptance testing phase of the division made by Khan and Khan (2012).

Ramler et al. (2014) divides the general architecture of an embedded system into three parts. In this classification the human machine interface (HMI) is the top layer. This is followed by the software running on the device and the lowest level are the hardware components of the machine which can be accessed through different analog and digital interfaces. As this master's thesis addresses only the acceptance testing of one instance of an embedded system and as it only has to verify whether the system fulfills its acceptance testing requirements, these abstraction levels can be overlooked. System under test (SUT) can be viewed at a level where only the inputs and the outputs of the system are considered important. Also for this purpose, the black box testing methodology seems to be the appropriate testing manner.

Acceptance testing of a payment terminal software can be seen as a testing phase where the UI of the device and the use cases of the device are tested at the final production level, i.e. through using the real buttons of the device under test and observing that the expected messages can be seen through the screen of the same device. This can be seen as an effort to automate a real human user using the payment terminal.

2.4 Test Suite Syntax

Test suite syntax plays a significant role in an automated acceptance testing environment of payment terminals in terms of test readability, reusability and adaptivity. When building an automated acceptance testing environment, the tests should be understandable enough that the whole development team and all of the project's stakeholders can easily adopt to the test syntax.

According to the well recognized guidelines of test automation by Bach

(1996), test automation and the process that it automates should be kept carefully separated. Test automation should be built in a form that it is easy to review and distinct from the process that it automates. These guidelines should be taken into account also when determining a suitable test framework and test suite syntax.

When evaluating suitable test automation frameworks, it should be recognized that simplicity is a key factor of successful test automation. Software projects usually involve some sort of quality assurance (QA) or even a separated QA team. Projects also tend to involve fair amount of people with no technical background or programming skills and yet their responsibilities can still involve guaranteeing the quality of the software. Mosley and Posey (2002) recognize that high level test languages help to share the knowledge amongst the people that are responsible for the product. Sharing information and knowledge amongst the project's stakeholders helps achieving the objectives of test automation and builds up the morale amongst the people that are involved.

Lowell and Stell-Smith (2003) state that acceptance tests should be easy as possible to write or otherwise people working with the project will not write the tests as the task is seen unpleasant. In order to cope with changing requirements or updated features, the tests should be easy to maintain as people have to be able to update them even if they have been written by someone else. For this reason, the test cases should be human readable and understandable also to non-technical people. Test steps should be self explanatory and unambiguous.

Test cases in acceptance testing of a payment terminal contain relatively high amount of repetition, for example, test step of inserting a personal identification number (PIN) code is the same whether right or wrong PIN code is inserted or whether the test case would validate a credit or debit payment. For this reason, test case syntax should be as modular as possible in order to allow reuse of keywords with different parameters. Easily reusable keywords also allows fast creation of new test cases.

Tests can essentially be written in some conventional programming language, for example Java or Python, or by using some higher level language. There are many widely used test frameworks available for conventional programming languages, for example jUnit for Java (JUnit, n.d.). This, however, requires programming experience to some extent in order to be able to understand and modify existing tests or write new ones. This would mean the usage of conventional programming language would be opposing the guideline for writing the tests as understandable as possible and therefore it would be opposing the best practices of automated acceptance testing. On the other hand, test case syntax must be versatile enough to accommodate different kinds of testing scenarios and needs. Efficient use of variables must be possible and for example use of different kinds of loop structures must be supported. This leads to a situation where the abstraction level of the test cases has to be considered carefully.

```
@Test
public void validLogin() throws Exception {
    try {
        openBrowser();
        inputUsername("demo");
        inputPassword("mode");
        clickElement(By.name("Submit"));
        pageTitleShouldBe("Welcome")
    } catch (Exception e) {
        log.error(getClass().getName() + " failed. Exception:", e);
        takeScreenshot(getClass().getName() + "_failed");
        System.out.println(driver.getPageSource());
        throw e;
    }
}
```

Figure 2.3: Example of a jUnit test case that tries to login to website.

In addition to the test frameworks utilizing the use of some conventional programming language for test cases, there are also couple of well-recognized tests frameworks available that use a more natural language for writing the tests. These frameworks usually use the same libraries for interacting with the system under test as more low-level frameworks, but they allow a higher-level syntax in the actual test scripts. One popular example of this kind of higher level test framework is Cucumber (Cucumber, n.d.). Cucumber is an open source acceptance test framework that utilizes behavior-driven development (BDD) style. Cucumber uses Gherkin language that is designed to be human readable without previous knowledge of programming (Gherkin, n.d.). This means that also non-technical personnel involved with the project can understand the test cases.

```
Scenario: Valid login to webpage
  Given that I am on the homepage
  And I have typed credentials
  When I click on link Submit
  Then the page title should be "Welcome"
```

Figure 2.4: Example of a simple Cucumber test scenario and use of Gherkin language.

Another good example of a higher level test framework is Robot Framework (RF). RF is a generic keyword-driven test automation framework that allows creation of human readable test cases (Robot Framework, n.d.). Reusability and extendibility of high-level keywords is also made relatively easy (Stresnjak and Hocenski, 2011). Robot Framework User Guide (2015) also outlines that RF has a highly modular software architecture allowing it to be easily connected to any kind of SUT by using different test libraries.

Example of a Robot Framework test case can be seen in Figure 2.5 below. It is easy to see the intended test case execution by looking at the test case. This will be the goal for the environment proposed later on in this master's thesis.

```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.txt

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Figure 2.5: Example of a simple Robot Framework test suite (Robot Framework, n.d.).

Chapter 3

Proposed Architecture

Based on the requirements pointed out on Chapter 2, this part of the master's thesis will present an architecture for automated acceptance testing environment for payment terminal software. Components of the environment can be divided into hardware and software components and this Chapter is divided to sections accordingly.

In order to automate the acceptance testing of the payment terminals, test environment that can manipulate and observe the device through physical means has to be created. In other words, environment has to have some sort of a robot for pressing the buttons and screen of the device has to be observed. All this must be also controlled by some kind of combination of software.

Motivation for this research came from a payment terminal software provider as they needed a cost-efficient and simple automated acceptance test environment in order to lower the costs and speed up the acceptance testing process of their software development. Costs of automated acceptance testing can be divided into three parts: environment costs, costs of creating new tests and maintenance costs (Laapas et al., 2014). This Chapter will present an automated acceptance testing environment that is intended to minimize the costs of each part of this division.

Eficode Oy took responsibility of implementing the system according to the best practices of the industry. This proposal was initial plan for the project and it will be presented in this Chapter.

3.1 Overview

When planning an automated acceptance test environment for payment terminal software, environment has to be highly adaptive for different types of hardware and software features of different payment terminal models. This proposal was done for one payment terminal software provider who had several different models of payment terminals and altogether over 50 different software configurations for those devices.

Security is a top priority of payment terminal electronics and software. Therefore, it is not possible to access internals of the payment terminal hardware. This means that AAT environment has to be able to manipulate the physical interface of the device. This also creates requirement for supporting different types of keyboard layouts and screen locations. In other words, environment cannot be dependent of single manufacturer or payment terminal model.

One of the requirements for the AAT environment was also usage of open source technologies. For the reasons pointed out in Section 2.1, customer wanted that the environment is as open as possible. This also creates reputation and visibility regarding the security matters.

Other requirements for the AAT environment was simplicity, low cost, low need for maintenance and ability to run the tests continuously around the clock.

3.2 Hardware

Hardware for this proposal was intendedly kept simple and low-cost as possible. This proposal presents the use of just one Raspberry Pi 2 Mode B (Raspberry Pi 2, n.d.) computer as a main computer for AAT environment. Raspberry Pi 2 is proposed as it offers sufficient computing power for this project with low purchasing costs and can run a full Linux operating system. It is also small-sized and does not require any cooling equipment. Therefore, it suites well to this project as it can be situated easily to the environment and can be run continuously around the clock without concerns about wearing cooling fans for example.

3.2.1 The Robot

As internal electronics of the payment terminals are not accessible for security reasons, a robot is needed to be able to manipulate the physical UI of the payment terminals. The robot should therefore be able to accommodate different types of payment terminals and be able to press all types of buttons. Low cost and low need for maintenance are also requirements for this robot, as required by the customer. The robot should also be able to manipulate multiple payment terminals at the same time in order to allow parallel execution of acceptance tests. This is intended to reduce the time acceptance testing process takes overall, as the same tests have to be run on different models of payment terminals. Other option would be to make the changing of the device under test easy and fast so that the manual work required can be minimized.

One of the options for automating the pressing of the buttons of the payment terminals would be to manufacture a frame on top the payment terminal which would have actuators for pressing each button. This would allow quick entering of key sequences and simultaneous pressing of multiple buttons. Hobby-grade servo motors could be used as actuators in order to make this solution affordable. However, in order to support different kind of keyboard layouts and different sized payment terminals, the solution would require advanced mechanical engineering and thus the price of this solution could rise to become cost-ineffective for the customer. For these reasons, this option for payment terminal manipulator was not chosen.

Other option for automatically pressing the buttons of the payment terminals would be utilizing the use of robotic arm. A robotic arm would be able to emulate a human user accurately and depending on the used robotic arm, simultaneous pressing of the buttons could also be possible. Drawback on the use a robotic arm is the relatively high purchasing price of accurate and powerful robotic arms. This could be overcame by manufacturing the robotic arm with own resources and using some openly available plans (BCN3D-Moveo, n.d.) but this would require extensive use of time for building the arm from bottom up. For these reasons, the use robotic arm was not chosen.

Third option for automating the key strokes of the payment terminal would be to use a cartesian coordinate robot (CCR). Cartesian coordinate robot is a robot whose axis of control are linear and are perpendicular to each other (Costa, 1995). For pressing of one key of the payment terminal at a time would need a cartesian coordinate robot with at least three degrees of freedom allowing the robot to move in three dimensional space. For the scope of this project, this would be enough as it is only required to press one button of the payment terminal at a time. CCR would also be easily able to adapt to different kinds of keyboard layouts and payment terminal sizes as it can travel across any coordinates within its workspace. By choosing a CCR with a right-sized work space, it could be also possible to accommodate multiple payment terminals to the workspace at the same time. This would allow the execution of parallel acceptance tests within several different devices at the same time. For these reasons, the use of a CCR for manipulating the buttons of the payment terminals was chosen.

The master's thesis proposes the use of ShapeOko 2 3-axis Computer Numerical Control (CNC) milling machine (ShapeOko 2, n.d.) to be used as a manipulator. Even though the machine is intended for milling purposes, it can be turned into a cartesian coordinate robot when milling tool is removed. As ShapeOko 2 is a CCR with horizontal member supported at both ends, it can be also referred as a gantry robot as it resembles a gantry crane.

ShapeOko 2 is an open-source hardware project and plans of the machine are openly available on their GitHub (ShapeOko 2 Github, n.d.). This allows easy modifications to the hardware parts of the robot if needed.

ShapeOko 2 is controlled by an Arduino board running a program called GRBL (GRBL, n.d.). Controlling program is an open-source, high-performance G-code interpreter and it is used for controlling CNC milling machines in general (ShapeOko 2, n.d.). G-code commands are sent from Raspberry Pi 2 to the Arduino on the robot using serial communication.

Robot should be equipped with a pushing tool that can be manipulate the buttons. Pushing tool can be easily manufactured using for example 3D-printing techniques. Design of the pushing tool can be seen in Figure 4.2

3.2.2 Computer Vision Hardware

In order to automate human interaction with the payment terminals, AAT environment has to be able to observe the changing content on the screen of the payment terminal. As stated earlier, internal electronics are not accessible due to the security measures and this disallows for example the possibly to intercept the LCD communication line of the payment terminal in order to retrieve the image on the screen programmatically.

Therefore, AAT environment also requires computer vision as changes on the screen have to be observed visually. Manufacturer of Raspberry Pi offers low-price solution for this as a form of Raspberry Pi Camera Module (Raspberry Pi Camera Module, n.d.). This module was chosen for use in computer vision tasks of the AAT environment.

As the size and the location of the display differs between different models of payment terminals, optical hardware has to be able to adapt to different kinds of imaging circumstances. As it is proposed that working area of the robot could be equipped with several payment terminals at the same time, also the displays of the payment terminals have to be able to be read regardless of the number of the devices under test.

One solution for this could be equipping the AAT environment with multiple stationary cameras, more precisely one camera per each device under test. If the cameras would be stationary, this would create boundaries for the location and the size of payment terminal displays depending on the location of the optical hardware. Cartesian coordinate robot proposed also has a rigid structure moving on top of the devices under test and this could cause blocking of the visual contact between camera and the display of the payment terminal.

Other solution would be having a moving camera that could be driven to a needed location in order to perform machine vision tasks. Location of the display could be configured regarding to the payment terminal model and this solution would adapt easily for different kinds of display layouts. Moving of the camera equipment can be achieved easily by attaching the camera directly to the robot. This will however exclude the ability to simultaneously pressing the buttons and reading the screen as robot has to be driven to certain position for capturing the image from the display. Regardless of this limitation, this solution was chosen. More precisely, the camera was situated to the Z-axis assembly of the robot to the other side in respect to the pushing tool. This would minimize the required transitions when changing from pressing the buttons to capturing the images as the displays are typically located on top the numeric keypads on the payment terminals.

3.2.3 Card Feeder

In addition to the manipulation of the payment terminal buttons, also the card feeding functionality has to be automated. One option to accomplish this functionality would be using the ShapeOko 2 robot for inserting and removing the card from the payment terminal. This would require an attachment to the payment card in order to make the manipulation of the card possible with the same tool that is used to push the buttons of the devices under test. The AAT environment software would also require some kind of reset functionality in case the software would crash and the position of the card would be lost. Manipulation of the payment cards with the ShapeOko 2 robot would also make overall testing process slower as it would not be possible to press the buttons while inserting or removing the payment card to or from the payment terminal.

Other option would be manufacturing generally adaptable card feeders that could be used with different kinds of payment terminals. This solution would allow simultaneously inserting and removing of the payment card while manipulating the buttons with the robot. Advantage of this solution would also be that card feeders could know their state even if the software would crash as well as the reset functionality would be more simple to implement.

As insertion and removal of the credit card might be hard to accomplish in a simple way using just the robot described in previous section. This work proposes the use of generally designed card feeders to accomplish this task. Proposed card feeders consist of 3D-printed base plate that attaches to the payment terminal, servo motor and 3D-printed tray that attaches to the servo and to the credit card. Design of the card feeder can be seen in Figure 4.5

Card feeders were designed in a way that they can be used with any types payment terminals that have the card slot at the bottom side of the device. Standard hobby servos were used as servo motors in order to keep the cost of the setup low.

Arduino board will be used to drive the servos as it can easily provide the needed pulse width modulated (PWM) signal for the servos. Arduino is suggested in order to ensure quality and accuracy of the PWM signal compared to what can be produced easily with non-real-time operating system running on the Raspberry Pi. Raspberry Pi on the robot will communicate with Arduino through serial communication.

3.3 Software

As stated in the section 2.4, automated acceptance tests should be simple and understandable enough to actually make the automated testing efficient and beneficial. Open source solutions should be favored as this was requested by the customer and to achieve benefits described in the Section 2.1. For these reasons, software decisions of the AAT environment should be carefully considered in order to achieve good maintainability, compatibility and overall simplicity.

For software part of this AAT environment, Raspbian Wheezy is proposed for the operating system. Raspbian is the official supported operating system for Raspberry Pi by Raspberry Pi Foundation (Raspbian, n.d.). Raspbian is based on widely-used Debian Unix-like operating system. This allows the use of components developed for Debian to be used with this AAT environment.

3.3.1 Test Framework

Based on the guidelines and comparison presented in Section 2.4, the choice for test framework was considered in order to achieve the best usability, versatility and functionality. In order to maximize these measures, Robot Framework was chosen for the test framework. RF is an open-source, generic, keyword-driven test automation framework that has human readable test case syntax (Robot Framework User Guide, 2015), (Robot Framework, n.d.).

Robot Framework also has highly modular software architecture (Robot Framework User Guide, 2015) which allows the framework to be used with variety of testing libraries to connect to the system under test. This feature can be seen as a great advantage when implementing test libraries for machine control and computer vision. Illustration of this modular architecture can be seen in Figure 3.1 below.

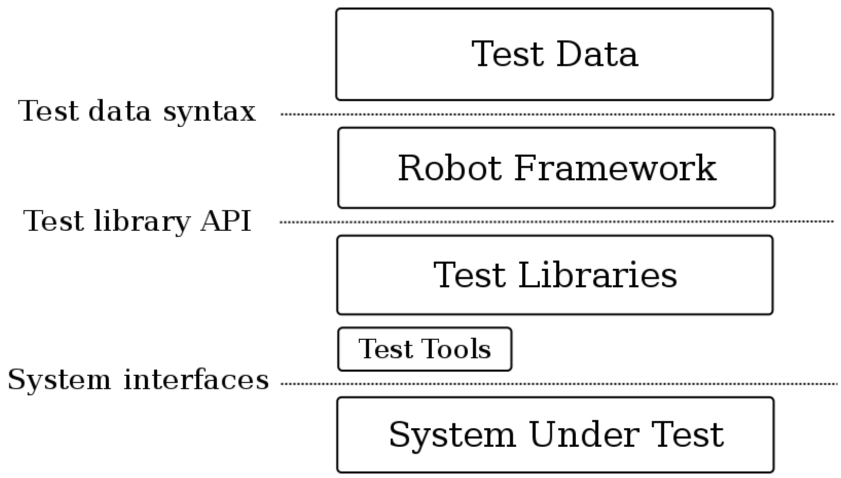


Figure 3.1: Illustration of modular software architecture of Robot Framework (Robot Framework software architecture, n.d.).

When RF tests are being executed, it generates clear report and log files of the test case execution results (Robot Framework User Guide, 2015). These files offer high level view of all test cases and step-by-step descriptions of individual test cases in order to make the debugging more easy.

Example of a test case can be seen in Figure 3.2. This test case describes automated RF acceptance test for entering invalid PIN code when trying to execute card purchase.

```
*** Settings ***

Documentation        A test suite for testing the machine movement and OCR
Resource             resource.txt
Test Teardown        Run Keywords    Remove Card    Go Home And Close Connection

*** Variables ***

${DEVICE_NUMBER}     1

*** Test Cases ***

Test Run For Device 1
    Set Home And Initialize    ${DEVICE_NUMBER}
    Press    red
    Screen Should Contain Text    Terminal ready
    Press    2    0    0    0
    Screen Should Contain Text    20,00
    Press    green
    Screen Should Contain Text    card
    Insert Card
    Press    1    4    5    2
    Press    green
    Screen Should Contain Text    failed
    Remove Card
    Screen Should Contain Text    Terminal ready

*** Keywords ***

Insert Card
    Card In    ${DEVICE_NUMBER}

Remove Card
    Card Out    ${DEVICE_NUMBER}
```

Figure 3.2: Example test case for invalid PIN code test

3.3.2 Test Libraries

As can be seen on Figure 3.1, RF requires external libraries to connect to the system under test. In the case of this AAT environment, those libraries would be a library for machine control, a library for computer vision and a library for card feeder manipulation. All these libraries can be written using

Python programming language that is supported out of the box by Robot Framework (Robot Framework, n.d.).

For machine control library, the environment has to be able to send G-code commands through USB serial communication to Arduino on the robot. For this, pySerial Python library is proposed as it includes implementation of the needed serial communication functionalities (pySerial, n.d.).

For the computer vision tasks of the environment, textual messages on the display are usually those that need to be verified. For this, character recognition is needed. Open source optical character recognition (OCR) engine called Tesseract OCR was chosen (Tesseract OCR, n.d.). It was initially developed by HP but since 2006 it has been developed by Google. In order to use Tesseract OCR with Python, a library named pytesseract was used (Pytesseract, n.d.).

Library for controlling the card feeders is the most simplest one of these three libraries. For this, pySerial Python library was also chosen to send the serial communication command to the Arduino controlling the card feeders. Library will handle sending of control commands to the Arduino controlling the card feeder servo motors.

Chapter 4

Results and Evaluation

This Chapter covers the subsystems and steps taken that were needed to achieve the testing environment described in Chapter 3. This Chapter first discusses the arrangements related to the hardware of the framework and then software related arrangements are presented and described. After presenting the built AAT environment, achieved results are discussed and finally the test environment presented in this thesis is evaluated based on whether it fulfilled the requirement of automating the acceptance testing of payment terminals set by the customer.

4.1 Hardware Arrangements

AAT environment presented in this master's thesis consists of several different hardware components. Environment had to be a smooth combination of manipulation and computing hardware. The hardware architecture is thought to be modular in the sense that every component has a specific functionality.

This allows easy maintenance and upgrade of each subsystem.

As stated in Chapter 3, one of the requirements for this AAT environment was affordable price. For this reason, hardware decisions have been made taking quality/price-ratio into consideration and hobby-grade electronics were used widely throughout the environment. 3D-printing was also utilized as a manufacturing technique of custom-made components for its relatively low manufacturing price and acceptable quality of outputted plastic parts.

Main components of the AAT environment are the robot that handles the manipulation of the payment terminals, Raspberry Pi 2 Model B single-board computer which is used as a main computer of the environment, two Arduino Uno boards for more specific control needs of certain components, camera for machine vision and 3D-printed payment card feeders for the payment terminals. These subsystems and components are described in following sections.

4.1.1 The Robot

As suggested in section 3.2.1, ShapeOko 2 open source 3-axis CNC milling machine was used as the robot manipulating the payment terminals. ShapeOko was built according to the instructions found from the homepage of the project (ShapeOko 2, n.d.). Construction was altered only regarding to the tool that was used as the spindle motor was substituted by 3D-printed pushing tool.

ShapeOko 2 has a working area of about 300 mm x 300 mm x 60 mm which means that it can accommodate up to three payment terminals at same time in the working area. This allows parallel test case execution i.e. test cases can be run at the same time with different terminals. Arrangement of the devices was implemented by dividing the work area into three sections. Each payment terminal was attached to a standard sized MDF-plate and each section of the working are can accommodate one of these MDF-plates. Holes were drilled into the working area and nuts were inserted into these holes at the back of the work bench. MDF-plates attach to these holes with screws enabling easy installation and removal of plates with different models of payment terminals. MDF-plates can be seen in Figure 4.6.

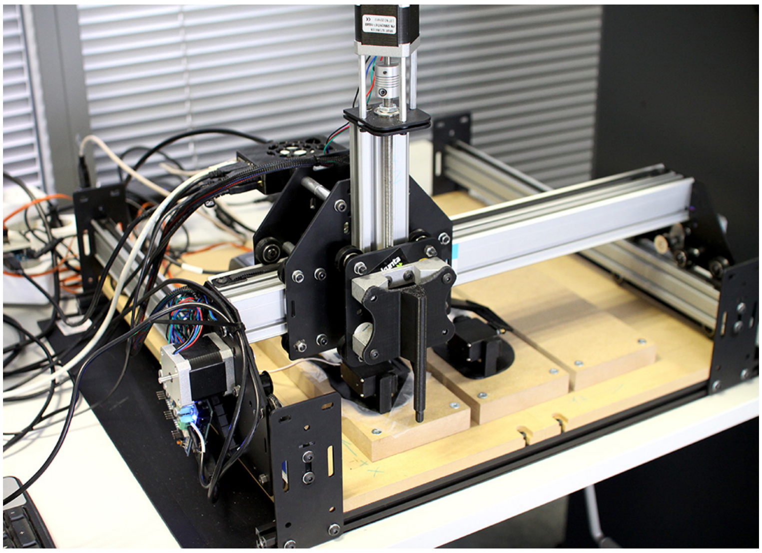


Figure 4.1: Robot in its production state.

Each axis of the robot is controlled by stepper motors. Use of stepper motors instead of servo motors offers affordable way of controlling each axis in a relatively fast and reliable manner. X- and Z- axises are both manipulated using one stepper motor on each axle and bigger Y-axis is manipulated using two parallel stepper motors. Manipulation of payment terminal buttons stresses the machine much less than actual milling of materials that the machine is designed for and, allowing faster movement of the machine that would be possible when executing actual milling job.

The robot was controlled using G-code that was sent from the main computer to an Arduino Uno attached to the robot. Arduino Uno and the main computer were connected via USB connection. More detailed description of the electronics can be found from section 4.1.2.

Section 3.2.1 suggested equipping the robot with a pushing tool and this was implemented to the final solution by 3D-printing the tool from PLA plastic. Tool consisted of two parts: cylindrical beam and a stem inside of it. Stem slides inside the beam and the two parts are separated with a spring. Spring provides the needed attenuation in order to forgive slight misalignments and too long trajectories when pushing the buttons of the payment terminals. Pushing tool can be observed in Figure 4.2 and Figure 4.1.

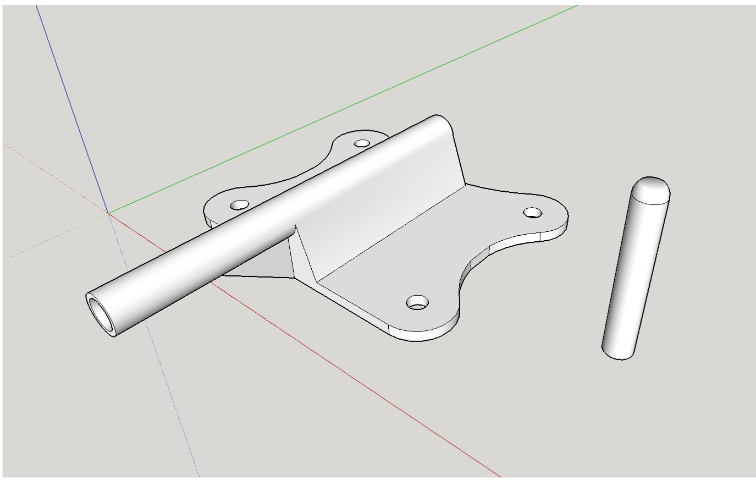


Figure 4.2: CAD design of the pushing tool. Metal spring is inserted inside to the cylinder and the stem on the right side of the image slides to the cylinder.

4.1.2 Computing Hardware

Raspberry Pi 2 Model B single-board computer is used as the main computer of the AAT environment. Raspberry Pi provides optimal computing power compared to it's price and has big community of users and developers world wide. 3D-printed enclosure was manufactured to protect the computer board and it was attached to the moving Z-axis assembly of the robot.

In addition to the Raspberry Pi 2, the robot also has two Arduino Uno boards for handling some specific functionalities of the AAT environment. One Arduino Uno is interpreting the G-code commands sent from the Raspberry Pi and it is connected to the stepper motors of the robot through a stepper motor driver shield (grblShield, n.d.).

Second Arduino Uno is handling the servo motor control of the card feeders. It is connected to the Raspberry Pi via USB connection and control commands to Arduino Uno are sent using serial communication. Arduino Uno board provides PWM signal to the servo motors and can accommodate three card feeders at the same time. Self-made circuit board was fabricated and attached on top of the Arduino Uno board in order to make connecting the servo motor cables easy.

Connection diagram and main electronic components are visualized in Figure 4.3.

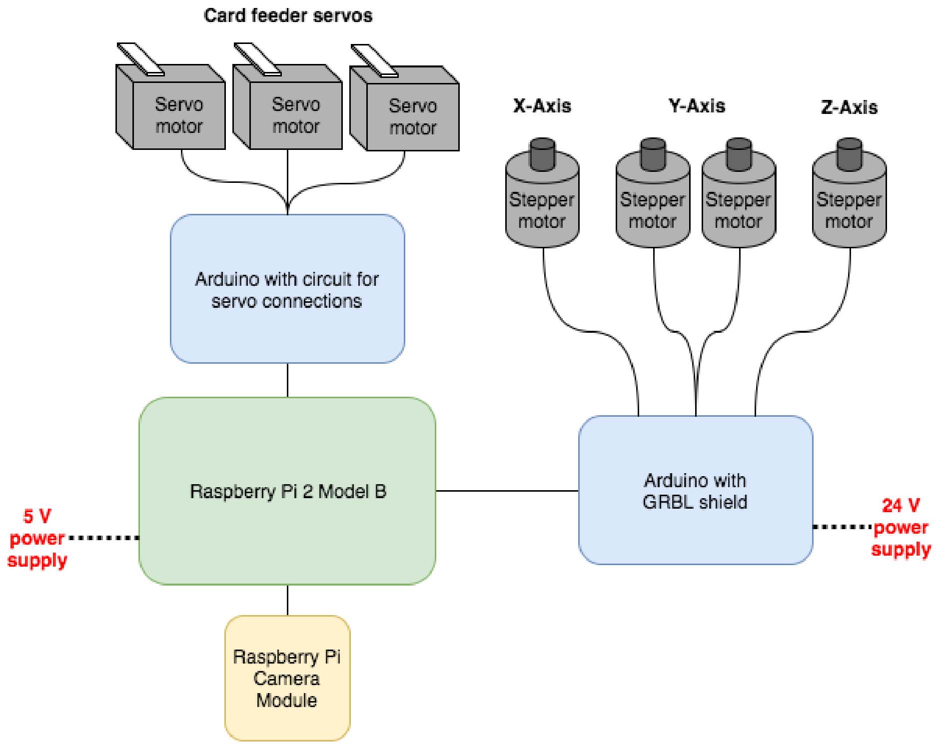


Figure 4.3: Main electronic components and connection diagram of the robot. Note that Y-axis is manipulated using two stepper motors.

4.1.3 Camera Arrangements

As suggested in section 3.2.2, Raspberry Pi's own camera module was used for machine vision hardware. Camera was attached to the bottom of the Raspberry Pi's enclosure and the enclosure was attached to the Z-axis assembly of the robot to the opposite side where the pushing tool is located. Camera can be moved within the X- and Y-axis while Z-axis movement of the camera isn't possible. Depth of focus of the camera provides clear image of the screen even when the distance between the lens and the screen differs slightly between different payment terminal models. Camera attachment can be seen in Figure 4.4.

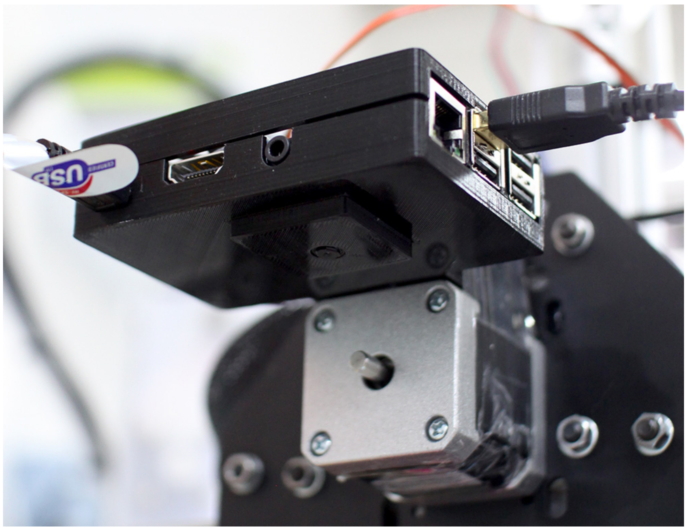


Figure 4.4: Camera is attached to the bottom of the Raspberry Pi's enclosure. Image also shows the attachment of the Raspberry Pi enclosure to the Z-axis assembly of the robot.

4.1.4 Card Feeder Arrangements

As suggested in section 3.2.3, card feeder structures were 3D-printed using PLA plastic. Finalized card feeders consist of bottom plate, payment card holder and servo motor. Servo motor attaches directly to the bottom plate and card holder attaches to the arm of the servo motor.

Simplistic design can be used with different kinds of payment terminals which have the card slot at the bottom side of the device. Flexibility provided by the plastic structure and the payment card itself allows the solution to be compatible with most of the payment terminals of this type. Design of the card feeders is presented in Figure 4.5. Figure 4.6 shows manufactured part installed to the environment presenting the servo installation and attachment of the card holder to the servo arm.

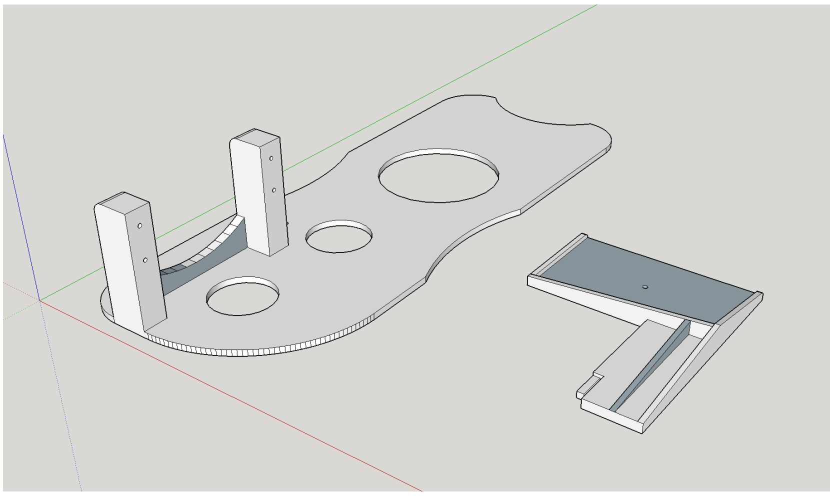


Figure 4.5: CAD design of the card feeder. Servo motor attaches to the bigger plate on the left and card holder on the right attaches to the arm of the servo motor. Card holder is designed to fit standard sized payment card.

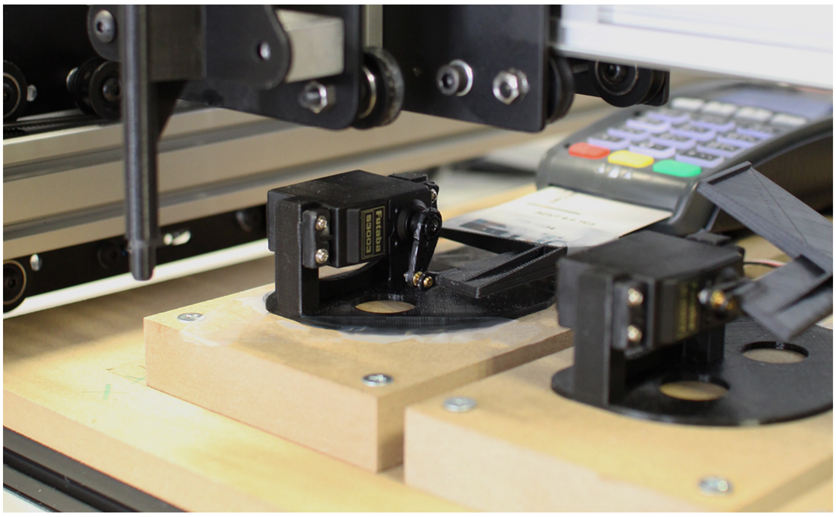


Figure 4.6: Card feeder installed to the environment. Image also presents the idea of MDF-plates described in section 4.1.1.

4.2 Software Arrangements

As proposed in section 3.3, this Chapter describes the decisions and arrangements regarding to the software point of view of the AAT environment. The initial proposal was followed rather loyally though some additional arrangements had to be implemented to the environment in order to increase usability and effectiveness.

The software architecture was implemented in a modular way in order to support the modularity of the hardware design. Implementation only included open source or self-made software components from the operating system to individual software libraries used in the AAT environment.

This section describes the individual software components of the AAT environment and their usage and function in the whole system. System configuration, test framework and libraries and the final test suite syntax are presented.

4.2.1 Software Architecture

As suggested in the section 3.3, Raspbian Wheesy Debian-based operating system was used with the Rasbperry Pi 2 Model B single-board computer. Operating system was used to run the test framework, test libraries and other software components and to handle the communication with different subsystems of the AAT environment.

Robot Framework was used as a test framework for its modularity, simplicity and versatility. RF was run on top of Python runtime environment and all test libraries were written using Python programming language (Python, n.d.). Python test libraries were implemented to handle the needed serial communication with the Arduino board on ShapeOko 2 and to the other Arduino board used for controlling the card feeder servo motors. Picamera (Picamera, n.d.) Python library was used for providing the needed Python interface for communication with the Raspberry Pi camera module. Overall visualization of the software architecture can be observed in Figure 4.7.

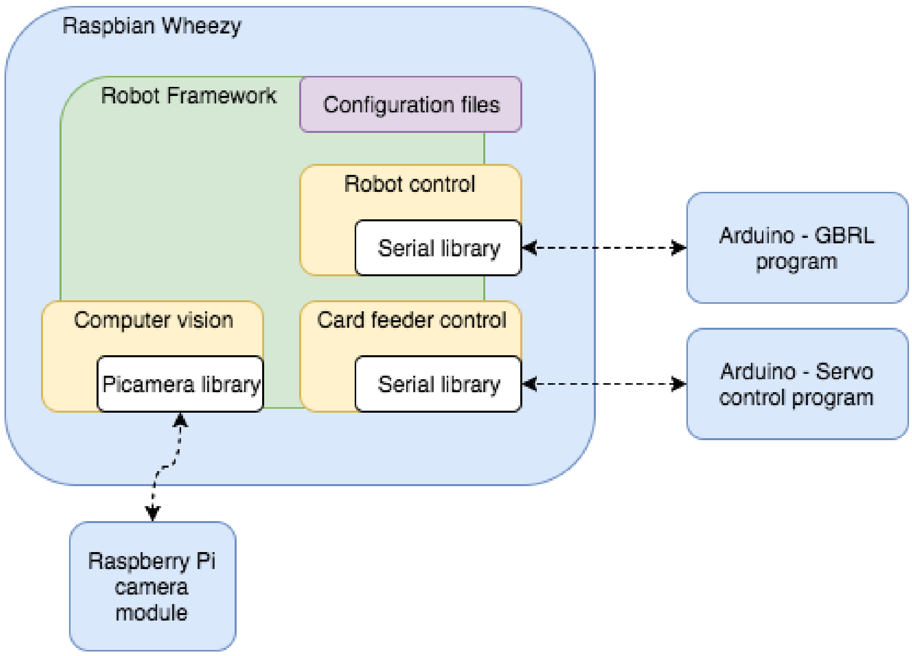


Figure 4.7: Software architecture of the AAT environment.

As different keyboard layouts have to be supported, configuration files for keyboard layouts were implemented. There are two types of configuration files: one for device locations in the working area of the robot and one for each keyboard layout. Configuration file for device locations defines the coordinates of "number one"-button and the height in respect of Z-axis where the transitions over the buttons are safe. This is Z-axis coordinate is used for transitions between pressings of buttons.

Configuration file for each keyboard layout defines the button locations in respect to the "number one"-button. The Z-axis coordinates defined in this file define the distance from the safe transition height to the full press of the button. Location of the screen of each device is also defined in these configuration files and it is used for driving the robot to the optimal place for capturing the image of the display of the device under test.

By dividing the configuration files, easy modification and addition of new device configurations is enabled. Desired configurations can be also changed easily at the test case level. Examples of these configuration files can be observed in Figure 4.8 and in Figure 4.9.

```
0.0 258.0 24.0          Device 1 (x- and y-coordinates of the device and z-position for changing tool position)
100.0 258.0 24.0        Device 2
200.0 150.0 24.0        Device 3
```

Figure 4.8: Configuration file for device locations in the working area of the robot.

```
0.0 0.0 8.0             One (X- and Y-coordinates and Z-position for pressing button)
17.0 0.0 8.0            Two
38.0 0.0 8.0            Three
0.0 -13.0 7.0           Four
17.0 -13.0 7.0          Five
38.0 -13.0 7.0          Six
0.0 -28.0 6.0           Seven
17.0 -28.0 6.0          Eight
38.0 -28.0 6.0          Nine
0.0 -40.0 4.0           Star
17.0 -40.0 4.0          Zero
38.0 -40.0 4.0          Hashtag
0.0 -54.0 2.0           Red
17.0 -54.0 2.0          Yellow
38.0 -54.0 2.0          Green
30.0 -180.0             Camera
0.0 18.0 10.0           F1
10.0 18.0 10.0          F2
25.0 18.0 10.0          F3
43.0 18.0 10.0          F4
```

Figure 4.9: Example of a configuration file of a device keyboard layout.

4.2.2 Robot Framework Test Framework

As proposed in section 3.3.1, Robot Framework was used as the test framework for the AAT environment presented in this master's thesis. RF was equipped with several different test libraries to achieve the desired functionality of the AAT environment.

RF is a generic keyword-driven test framework and this means that the keywords used for different test steps can be defined at a desired level of abstraction. Lowest abstraction level would be that one keyword would handle only one library method and highest would be that one keyword would be responsible for the whole test case. This allows high versatility but also makes the developer responsible of writing test cases according to commonly accepted best practices. Test cases developed in the scope of this master's thesis were implemented to be as human readable as possible. Also the devision of the test cases into test steps was intended to be intuitive. This naturally depends on the person that is planning the test steps but it was attempted to make each test step as clear as possible.

Test cases and steps were also divided into different keywords in order to achieve reusability. According to Martin (2009) any code written should be as readable and understandable as possible and these directions were used as guidelines when the test cases were implemented. It is also advised that code should be written in highly modular manner and this was followed when the keywords were combined in different abstraction levels.

Keywords used in the test cases were defined in three different levels: test library keywords, shared keywords and test suite specific keywords. Test library keywords are the most low level keywords and implement the functionality between RF and SUT using different interfaces. These were written using Python language. Test suite keywords, on the other hand, are the most high level keywords. These are defined within the test suite files and are only used within the particular suite.

AAT environment also introduced a resource file for combining the keywords that were shared with different test suites. Abstraction level of keywords found from this file can be qualified as middle or high level. Resource file is also used for defining common test libraries between test suites and common set-up and tear-down commands of test cases and suites. Resource file is imported to each test suite file. Example of partial resource file can be seen in Figure 4.10

```
*** Settings ***

Documentation   A resource for payment terminal acceptance testing

Library     ../libs/StreamLibrary.py
Library     ../libs/OCRLibrary.py
Library     ../libs/ArduinoServoLibrary.py

*** Variables ***

${IMAGE_PATH}     ../ocr-images
${CONF_FILE_PATH}     ../conf

*** Keywords ***

Open Connection
    StreamLibrary.OpenConnection

Go Home And Close Connection
    Go To Home
    Lower Tool
    Close Connection
```

Figure 4.10: An example of a partial resource file for Robot Framework tests.

4.2.3 Robot Control and Card Feeder Libraries

For sending the control commands to the ShapeOko 2 robot with Robot Framework, a robot control library was implemented using Python language. Control commands for the robot are given using G-code commands and those are being sent using serial communication protocol. The library defines keywords that can be used within the test cases. As RF supports combining the low-level keywords into higher level keywords, the library keywords were implemented to be reasonably generic. This helped to keep the library as simple and as possible.

Desired G-code commands are being produced according to the configuration files described in section 4.2.1. Library reads the coordinates of the devices and different buttons and by combining these, forms the needed G-code command to drive the robot into particular location. Library has a *go_to()*-method which takes the button name as parameter to drive the robot into the desired position. *Press_button()*-method is used to press the button when to robot is reached the desired position on top of the button.

Library has methods for setting the home position which is used in the initialization phase of the library after it has been imported into a RF test suite. Library also implements methods for going into home position, going to the right position for image capture and individual methods for lowering and raising the pushing tool of the robot. These can be used as keywords within the test cases and they work in respect to the device locations.

For controlling the card feeder, another test library was also implemented using Python language. Control command for card feeder Arduino board are being sent using serial communication and library takes care of this interaction between the RF and the Arduino board. Card feeder library only implements an *update()*-method that takes the angle and the card feeder number as parameters. This method sends the control command to the Arduino board of the card feeders and can be used as a keyword from other RF keywords or test cases.

Based on the work done in this master's thesis, open source RF library was published (Robot Framework CNC Library, n.d.). This library can be used for easy controlling of devices that use serial communication as a communication protocol and are controlled using G-code commands. The library is intended for use in similar circumstances described in this master's thesis but can also be used as general G-code control library for Robot Framework.

4.2.4 Card Feeder Software

For controlling the servo motors of the card feeders with an Arduino board, an Arduino program was developed. As described in the previous section, the control commands are sent to the Arduino using serial communication. Messages read by Arduino consists of two parts: card feeder number and desired angle of the servo. After receiving the message, Arduino program interprets the device number and angle from it and drives the appropriate card feeder.

Servo motors of the card feeders are controlled using PWM control signals. The Arduino program can drive the servo motors to every angle that the servo motor is capable of moving and the angles of inserting the card and removing the card can be defined in the test case level.

4.2.5 Computer Vision Library

As proposed in section 3.3.2, computer vision library was implemented for extracting the optical features from the display to a format that can be interpreted programmatically. The main task of the computer vision library is to interpret the text displayed on the screen of the payment terminal.

Computer vision library was implemented using Python language and Tesseract optical character recognition (OCR) engine is used to extract the found characters to textual format. Image captured by the Raspberry Pi camera module is slightly manipulated in order to make the text extraction more efficient and reliable. Image manipulations are made using OpenCV Python library (OpenCV, n.d.). Image is first being slightly blurred using Gaussian blur filter in order to reduce the amount of disturbance caused by pixel edges of the display. Color space of the image is then converted to gray-scale. Finally, the gray-scale image is converted into binary BW-image by comparing the pixel value to a certain threshold value. The threshold value is adjusted according to the screen brightness and lightning conditions of the space where the robot is situated. These image manipulations produce an image where text in the display is clearly distinguishable from the other features providing good foundation for the character recognition.

Tesseract OCR engine can extract any kind of common characters from the image and this can sometimes cause unwanted noise as small dirt particles and disturbances in the image can be interpreted as some exotic special character. In order to make the task of OCR engine easier in addition to the binary BW-image, possible characters are white-listed. Final list of possible characters that are accepted by the OCR engine is: ABCDEFGHI-

JKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789,€. This also helps the OCR engine to distinguish right characters from possible similar looking foreign-language counterparts.

The computer vision library outputs all found textual features from the source image and the validation of the right content is being done using Robot Framework.

4.2.6 Test Syntax

Robot Framework files are divided into different parts that all have specific functionality and purpose. This helps to observe the different configurations and used keywords within the test suite in order to gain comprehension of the functionality of a particular file. As mentioned earlier, the project structure is divided into a shared resource file and individual test suite files. Same syntax applies to both kinds of files, only the scope of the definitions changes according to the type of the file. Resource file is divided into three sections: settings, variables and keywords. Test suite files are divided into four sections: settings, variables, test cases and keywords.

The settings section of the file defines all the needed settings for executing the test cases. This includes all resource and library imports and test setup and teardown definitions. In this project, all the library imports are done in the resource file which is imported to the test suites in their settings sections. Test setup and -teardowns are defined at test suite level.

Variables section defines all the used variables within the test cases. Variables section of the resource file are used for defining common variables and for example the used directory paths. In other words, this sections defines the location of the image directory used by the computer vision, the location of the configuration files of the device locations and the configuration files of different keyboard layouts of the payment terminals. Variables are defined using RF's ${variable} annotation.

Test cases section of the test suite files are used for defining all the test cases within the test suite and all the test steps included in the test cases. Test cases are defined by naming them in the first line and then defining the test steps by indenting the names of the used keywords with at least two space characters or one tab character under the name of the test case.

Keywords are defined in the same way as test cases. Each keyword definition begins with the name of the keyword followed by indented names of used lower level keywords or library methods. Keywords can be built modularly into different layers by using lower level keywords in higher level keyword definitions. Example test case can be seen in Figure 3.2.

4.2.7 Test Results

For the testing to be actually useful and informative, clear test reports and error descriptions have to be generated. Robot Framework is useful for this purpose as it generates by default three types of clear and easily understandable test result files after executing the test suite under examination. Two files are outputted in *.html*-format making it possible to examine the reports interactively using web browser. One file is also outputted in *.xml*-format making it convenient to integrate the test results into other testing tools. RF also supports generation of other types of reports out-of-the-box and it is possible to produce for example *xUnit*-styled report from the test execution. (Robot Framework User Guide, 2015)

Report.html-file can be user to review the overall status of executed test suites and cases. This report gives an overall view to the testing project and different outcomes of the tests are marked in bright colors. The green color represents passed test and red color represents failed test. Example of passed test report file can be seen in Figure 4.11 and example of failed test report can be seen in Figure 4.12.

Log.html-file contains more detailed representation of the test cases. Each test step is shown here and the internal keywords and library methods used by the keyword are layered under each test step. If test step fails during the test execution, the stack trace of that particular command is added to the log file and can be easily observed. Example of log file can be seen in Appendix A.

Robot Framework also allows tagging the test cases with different kinds of tags. Tags can be used to group different test cases for test execution and they can be also be used for marking the criticality of the test case. The overall result of the test execution is determined based on the passed critical tests. If any of the test cases that are marked with *critical* tag fails, the overall test execution is considered failed. In other words, if the test run contains tests with *critical* and *non-critical* tags, the non-critical tests can fail without having an effect to the overall result of the test execution. (Robot Framework User Guide, 2015)

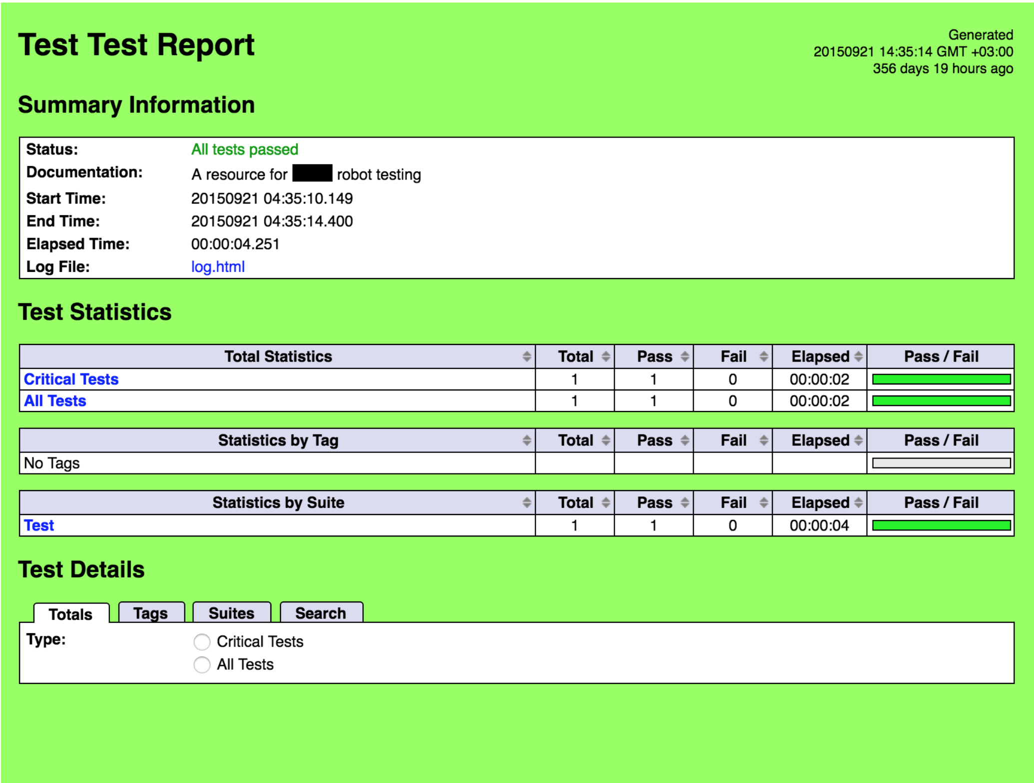
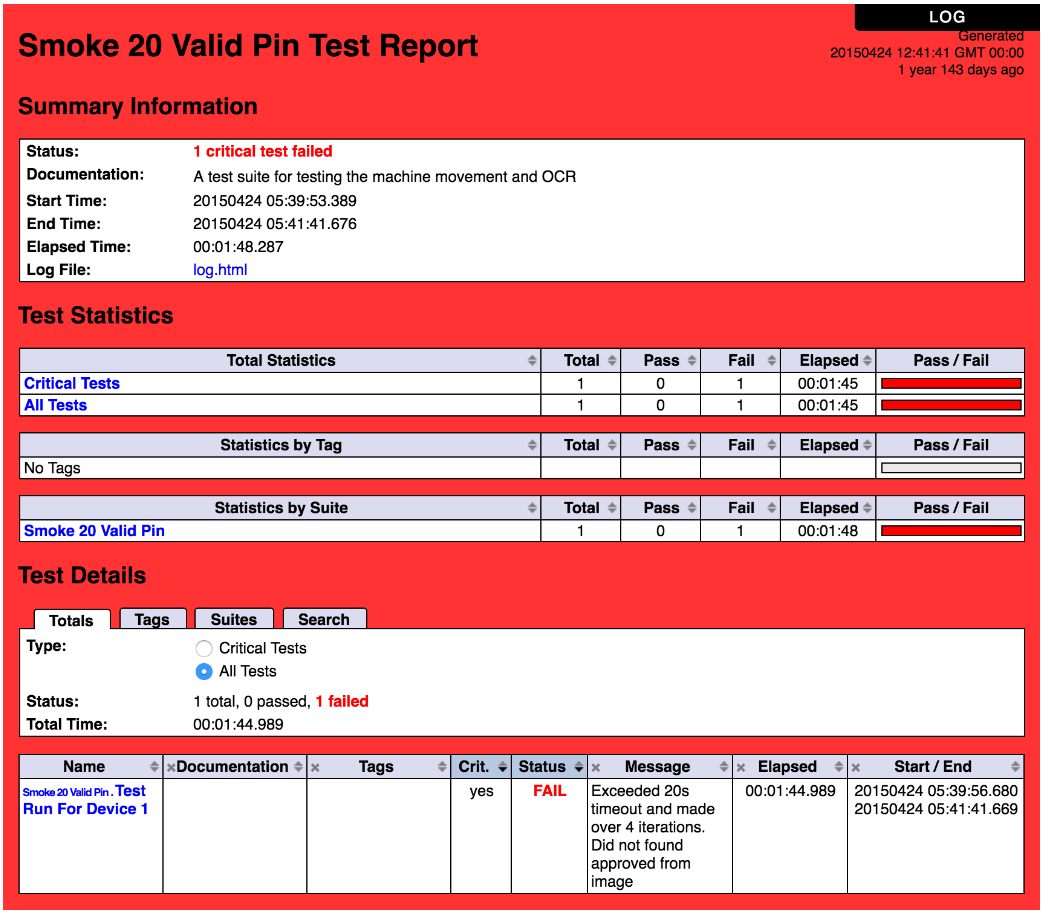


Figure 4.11: Report of passed tests.



LOG
Generated
20150424 12:41:41 GMT 00:00
1 year 143 days ago

Smoke 20 Valid Pin Test Report

Summary Information

Status:	1 critical test failed
Documentation:	A test suite for testing the machine movement and OCR
Start Time:	20150424 05:39:53.389
End Time:	20150424 05:41:41.676
Elapsed Time:	00:01:48.287
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	0	1	00:01:45	
All Tests	1	0	1	00:01:45	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Smoke 20 Valid Pin	1	0	1	00:01:48	

Test Details

Totals | Tags | Suites | Search

Type:	Critical Tests All Tests
Status:	1 total, 0 passed, 1 failed
Total Time:	00:01:44.989

Name	Documentation	Tags	Crit.	Status	Message	Elapsed	Start / End
Smoke 20 Valid Pin . Test Run For Device 1			yes	FAIL	Exceeded 20s timeout and made over 4 iterations. Did not found approved from image	00:01:44.989	20150424 05:39:56.680 20150424 05:41:41.669

Figure 4.12: Report of failed tests.

Chapter 5

Discussion

This Chapter will discuss the problem statements based on the proposal and implementation described in previous Chapters of this master's thesis. Possible future research topics around the subject matter of this thesis are presented related to the research questions.

5.1 Benefits of Open Source

Even though open-source license of a software does not guarantee the quality and excellence of the product compared to the closed source counterparts, the usage of open source solutions was beneficial for this project. As overall budget of the project was set rather low, open source products provided advantage over proprietary solutions.

Use of open source software provided possibility to evaluate the possible tools more throughly before actually taking them into the project. This was especially beneficial in terms of efficiently evaluating the potential tools and keeping the development time frame short.

Use of open source products also provided benefits to the customer organization whom the project was developed for. As the system under testing had strict security requirements, open source solutions provided visibility and transparency to the users and developers of the tested product.

Comparison between open source and proprietary products can be done in numerous different aspects and research done within this master's thesis was restricted due to time and cost limitations of the project. Future research could address this comparison more thoroughly by comparing AAT environments developed strictly with either open-source or proprietary components.

5.2 Characteristics of Payment Terminals

Different types of payment terminals were examined within this master's thesis and it was found out that due to the simple function of the payment terminal, the design usually involves few common parts: a display, a keypad and a card slot.

Scope of the master's thesis was limited to certain types of payment terminals and more exotic models were left out of consideration. Developed environment only supports payment terminals using chip card slot for inserting the payment card and use of other reading methods of the payment card, e.g. reading of magnetic stripe or NFC-chip, are not supported.

For the future work, possibility to support other reading methods of the payment card is suggested to being researched.

5.3 Approaches for Test Automation

Black box testing was used as a testing methodology within the work done in this master's thesis. Choice of the methodology was done entirely based on the definitions found from the literature around this topic. Use of black box testing methodology in the automated acceptance testing of embedded systems can be seen as most reasonable option as it imitates the final user most accurately. Other methodologies would have required more in-depth knowledge of the underlaying systems of the devices and this would not have emulated the final human user as accurately as black box testing methodology did in this case.

Methodology worked well in the AAT environment implemented in this master's thesis. AAT environment imitated final human user to the extent that it was possible to mostly automate the manual testing of the payment terminal, which was the goal of this project. As the AAT environment presented in this master's thesis concentrated on validating only the textual content of the payment terminal display, other visual validations were still left to be testes manually.

5.4 Syntax for Test Suites

Robot Framework was selected as a testing framework of the AAT environment. Choice of the framework and therefore also the test syntax was done based on literature review and examination of different tools. Robot Framework was selected for its modularity and versatile and human-readable test syntax. Use of RF proved to be robust and it was able to implement all the desired functionalities using the framework.

For future research, it is encouraged to arrange surveys and interviews related to the different acceptance testing frameworks. Research done within this master's thesis did not involve any investigation about current opinion atmosphere around the topic of acceptance testing tools used in testing of embedded software. This kind of research would be valuable to the future projects done in the field of automated acceptance testing. Multiple competitive testing tools exists and as the evaluation of the tools require extensive usage of different solutions, it would be beneficial if comparative and unbiased data would be widely available.

Chapter 6

Conclusions

This master's thesis presented a proposal and implementation of automated acceptance testing environment for payment terminal software and addressed the theories and problems related to the topic. Presented AAT environment was joint combination of open source hardware and software and was formed by the requirements of Eficode Oy's customer. AAT environment presented in this thesis was able to fulfill the requirement of automating a majority of the software testing of the payment terminals which was previously done manually.

Literature review of this master's thesis addressed the four problem statements introduced in the beginning of this master's thesis. Research questions were also discussed and suggestions for future research were presented in Chapter 5 after the implemented AAT was introduced in previous Chapters.

Presented architecture and testing solution proved to be adaptive to different kinds of payment terminals and also enabled testing of three different payment terminals in parallel setup to reduce the overall duration of the acceptance testing process. Solution also provided transparency to the users and developers of the security critical system under testing. As a result of this project, an open source Robot Framework library was also published for controlling any kinds of robots supporting serial communication and G-code commands.

This master's thesis also lays a promise of how commonly and inexpensively available components can be used in demanding applications. By combining different open source products, highly adaptive AAT environment was created for the needs of automated acceptance testing of payment terminal software successfully.

Bibliography

Roberto Acosta et al. *Open source hardware*. PhD thesis, Massachusetts Institute of Technology, 2009.

James Bach. Test automation snake oil. *Windows Tech Journal*, 10, 1996.

BCN3D-Moveo. URL `https://github.com/BCN3D/BCN3D-Moveo` Accessed 5.9.2016, n.d.

Bart Broekman and Edwin Notenboom. *Testing embedded software*. Pearson Education, 2003.

Larry J Costa. Three-axis cartesian robot, December 19 1995. US Patent 5,476,358.

Cucumber. URL `https://cucumber.io/` Accessed 7.9.2016, n.d.

Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, (4):42–52, 2009.

Gherkin. URL `https://github.com/cucumber/cucumber/wiki/Gherkin` Accessed 7.9.2016, n.d.

GRBL. URL `https://github.com/grbl/grbl` Accessed 15.7.2016, n.d.

grblShield. URL `https://www.synthetos.com/project/grblshield/` Accessed 1.9.2016, n.d.

Borge Haugset and Geir Kjetil Hanssen. Automated acceptance testing: A literature review and an industrial case study. In *Agile, 2008. AGILE'08. Conference*, pages 27–38. IEEE, 2008.

Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

Ingenico payment terminal. URL `http://www.netskauppa.fi/images/t/24-85-PrimaryImage.image.ashx` Accessed 4.4.2016, n.d.

JUnit. URL `http://junit.org/` Accessed 7.9.2016, n.d.

Mohd Ehmer Khan and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl*, 3(6), 2012.

Antti Laapas et al. Cost-benefit analysis of using test automation in the development of embedded software. 2014.

Charles Lowell and Jeremy Stell-Smith. Successful automation of gui driven acceptance testing. In *Extreme programming and agile processes in software engineering*, pages 331–333. Springer, 2003.

R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, 2009. ISBN 9780132350884. URL `https://books.google.com/books?id=hjEFCAAAQBAJ`.

Lorraine Morgan and Patrick Finnegan. Benefits and drawbacks of open source software: an exploratory study of secondary software firms. In *Open*

Source Development, Adoption and Innovation, pages 307–312. Springer, 2007.

D.J. Mosley and B.A. Posey. *Just Enough Software Test Automation*. Just enough series. Prentice Hall PTR, 2002. ISBN 9780130084682. URL `https://books.google.com/books?id=PEBvfWESIt4C`.

G.J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. IT-Pro collection. Wiley, 2011. ISBN 9781118133156. URL `https://books.google.fi/books?id=GjyEFPkMCwcC`.

Hung Q Nguyen. *Testing applications on the Web: Test planning for Internet-based systems*. John Wiley & Sons, 2001.

OpenCV. URL `http://opencv.org/` Accessed 18.7.2016, n.d.

James W Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. *Software Engineering, IEEE Transactions on*, 30(4):246–256, 2004.

Picamera. URL `http://picamera.readthedocs.io/en/release-1.12/` Accessed 7.9.2016, n.d.

R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill higher education. Boston, 2005. ISBN 9780073019338. URL `https://books.google.com/books?id=bL7QZHtWvaUC`.

pySerial. URL `http://pythonhosted.org/pyserial/` Accessed 4.5.2016, n.d.

Pytesseract. URL `https://pypi.python.org/pypi/pytesseract` Accessed 4.5.2016, n.d.

Python. URL `https://www.python.org/` Accessed 7.9.2016, n.d.

Rudolf Ramler, Werner Putschögl, and Dietmar Winkler. Automated testing of industrial automation software: Practical receipts and lessons learned. In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, MoSEMInA 2014, pages 7–16, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2851-7. doi: 10.1145/2593783.2593788. URL `http://doi.acm.org/10.1145/2593783.2593788`.

Raspberry Pi 2. URL `https://www.raspberrypi.org/products/raspberry-pi-2-model-b/` Accessed 1.6.2016, n.d.

Raspberry Pi Camera Module. URL `https://www.raspberrypi.org/products/camera-module/` Accessed 1.6.2016, n.d.

Raspbian. URL `https://www.raspberrypi.org/downloads/raspbian/` Accessed 4.5.2016, n.d.

Robot Framework. URL `http://robotframework.org/` Accessed 1.6.2016, n.d.

Robot Framework CNC Library. URL `https://github.com/Eficode/robotframework-cnclibrary` Accessed 1.8.2016, n.d.

Robot Framework software architecture. URL `http://robotframework.org/img/architecture-big.png` Accessed 1.6.2016, n.d.

Robot Framework User Guide. URL `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html/` Accessed 6.5.2016, 2015.

Erik Rubow. Open source hardware. Technical report, Citeseer, 2008.

ShapeOko 2. URL `http://www.shapeoko.com/wiki/index.php/ShapeOko_2` Accessed 1.6.2016, n.d.

ShapeOko 2 Github. URL https://github.com/shapeoko/Shapeoko_2 Accessed 15.7.2016, n.d.

I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011. ISBN 9780137053469. URL https://books.google.fi/books?id=l0egcQAACAAJ.

Stanislav Stresnjak and Zeljko Hocenski. Usage of robot framework in automation of functional test regression. In *ICSEA 2011 The Sixth International Conference on Software Engineering Advances*, 2011.

Tesseract OCR. URL https://github.com/tesseract-ocr/tesseract Accessed 4.5.2016, n.d.

Appendix A

First appendix

Smoke 20 Valid Pin Test Log

Generated
20150424 12:41:41 GMT 00:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	0	1	00:01:45	
All Tests	1	0	1	00:01:45	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Smoke 20 Valid Pin	1	0	1	00:01:48	

Test Execution Log

TEST SUITE: **Smoke 20 Valid Pin** 00:01:48.287

Full Name: Smoke 20 Valid Pin
Documentation: A test suite for testing the machine movement and OCR
Source: /home/pi/[redacted]/suites/smoke_20_valid_pin.txt
Start / End / Elapsed: 20150424 05:39:53.389 / 20150424 05:41:41.676 / 00:01:48.287
Status: 1 critical test, 0 passed, 1 failed
1 test total, 0 passed, 1 failed

TEST CASE: **Test Run For Device 1** 00:01:44.989

Full Name: Smoke 20 Valid Pin.Test Run For Device 1
Start / End / Elapsed: 20150424 05:39:56.680 / 20150424 05:41:41.669 / 00:01:44.989
Status: FAIL (critical)
Message: Exceeded 20s timeout and made over 4 iterations. Did not found approved from image

KEYWORD: **resource.Set Home And Initialize** 1 00:00:08.414
KEYWORD: **StreamLibrary.Press** red 00:00:05.189
KEYWORD: **resource.Screen Should Contain Text** Terminal ready 00:00:08.767
KEYWORD: **StreamLibrary.Press** 2, 0, 0 00:00:11.207
KEYWORD: **resource.Screen Should Contain Text** 2,00 00:00:09.234
KEYWORD: **StreamLibrary.Press** green 00:00:04.311
KEYWORD: **resource.Screen Should Contain Text** card 00:00:09.515
KEYWORD: **resource.Insert Card** 00:00:05.017
KEYWORD: **StreamLibrary.Press** 1, 2, 3, 4 00:00:12.735
KEYWORD: **StreamLibrary.Press** green 00:00:04.095
KEYWORD: **resource.Screen Should Contain Text** approved 00:00:21.373
Start / End / Elapsed: 20150424 05:41:15.206 / 20150424 05:41:36.579 / 00:00:21.373
KEYWORD: **StreamLibrary.Go To Camera** 00:00:00.905
KEYWORD: **OCRLibrary.OCR Find From Image** ${Text}, ${IMAGE_PATH} 00:00:20.455

Start / End / Elapsed: 20150424 05:41:16.121 / 20150424 05:41:36.576 / 00:00:20.455

```
05:41:36.516   INFO  Iteration: 1
                     End Image capture. 2015-04-24 12:41:17.735340
                     Start Image processing. 2015-04-24 12:41:17.735664
                     Stop Image processing. 2015-04-24 12:41:22.453540
                     Start Character recognition. 2015-04-24 12:41:22.453901
                     Transaction
                     declined
                       Remove card


                     Stop character recognition. 2015-04-24 12:41:24.257069
                     Stop OCR 2015-04-24 12:41:24.257288
                     Transaction
                     declined
                     e Remove card


                     Stop character recognition. 2015-04-24 12:41:25.969656
                     Stop OCR 2015-04-24 12:41:25.969887
                     Transaction
                     declined
                     T Remove card


                     Stop character recognition. 2015-04-24 12:41:27.874151
                     Stop OCR 2015-04-24 12:41:27.874445
                     Transaction
                     declined
                     T Remove card


                     Stop character recognition. 2015-04-24 12:41:29.668542
                     Stop OCR 2015-04-24 12:41:29.668771
                     Transaction
                     declined
                     T Remove card


                     Stop character recognition. 2015-04-24 12:41:31.455504
                     Stop OCR 2015-04-24 12:41:31.455761
                     Transaction
                     declined
                     T Remove card


                     Stop character recognition. 2015-04-24 12:41:33.155385
                     Stop OCR 2015-04-24 12:41:33.155639
                     Transaction
                     declined
                     T Remove card


                     Stop character recognition. 2015-04-24 12:41:34.833477
                     Stop OCR 2015-04-24 12:41:34.833708
                     Transaction
                     declined
                     T Remove card


                     Stop character recognition. 2015-04-24 12:41:36.514911
                     Stop OCR 2015-04-24 12:41:36.515141
05:41:36.520   FAIL  Exceeded 20s timeout and made over 4 iterations. Did not found approved from image
```

+ TEARDOWN: **BuiltIn.Run Keywords** Remove Card, Go Home And Close Connection 00:00:05.084