



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Visual GUI Testing of Embedded Systems

Master's thesis in Software Engineering

Maciej Makarewicz

Visual GUI Testing of Embedded Systems
Maciej Makarewicz

© Maciej Makarewicz, 2016

Chalmers University of Technology
SE 412 96 Göteborg
phone: +46 31 772 10 00
www.chalmers.se

Department of Computer Science and Engineering
Göteborg, 2016.

Abstract

The increasing demand on software quality drives emergence of new testing technologies. While well-known methods exist for automated graphical user interface (GUI) testing, such as coordinate-based, widget-based and visual GUI testing (VGT), not much has been done to automate GUI testing of software for embedded systems. This thesis approaches this problem by evaluating the applicability of VGT – a technology that analyses the bitmap contents of the system-under-test's (SUT) GUI to automatically drive tests.

The study was conducted in cooperation with a Gothenburg-based company that operates in the automotive field. A prestudy was conducted in order to recognise the company's needs when it comes to VGT of their embedded systems. A VGT framework was implemented. Furthermore, an open-source GUI automation tool was adapted to match the company's needs. Finally, experiments were conducted that compared the performance of the two resulting VGT frameworks when it comes to test-case (TC) development and TC maintenance costs.

The work concludes with guidelines on how to apply VGT for testing of software for embedded systems, with emphasis on how to minimise TC development and maintenance costs.

KEYWORDS: VGT, Visual GUI Testing, automation, embedded systems, test-case development, test-case maintenance, framework implementation, Sikuli.

Acknowledgements

I would like to thank Emil Alégroth for his feedback and patience while supervising the work carried out in this master thesis. Regarding the technical part, I would like to extend my gratitude to Robin Persson and Christopher Olofsson who have been supervisors at the company where the thesis was carried out.

I would also like to thank my parents, extended family and friends, especially Basia Gosek, for the continuous and unbreakable support given during the course of this work. I also want to thank Lisa Bailey for proofreading this report.

Maciej Makarewicz, Göteborg, 22nd of March 2016

Glossary

action Part of the HMI-FU data interface. A way for an FU to communicate with the HMI. Actions can carry any type of data or can be merely signals to HMIs. Can be compared to a function call that may carry data.

data In the context of HMI-FU communication: Part of the HMI-FU data interface. A way for the HMI to share data with FUs. The format of the data is arbitrary. Can be compared to a function call that is meant to carry data.

event Part of the HMI-FU data interface. A way for the HMI to communicate with FUs. A signal telling the FU that something has happened or that some action is required. Can be compared to a function call.

faceplate An area where the HMI can display simulated hardware controls such as buttons and rotary controls.

FU Functional Unit. A peripheral of the HMI. The HMI can send events, indications and data to an FU and it can also capture actions issued by an FU. In a car, a car radio or a parking brake are examples of what a typical FU could be.

FU Simulator FU Simulator, or functional units simulator is a subsystem of the Editor. It lets the developer simulate his/her HMI by changing FUs' indications and data that they present to the HMI. They also let the developer send events from FUs to the HMI. This can be done either manually or using scripts.

fusdk A library that makes communication with FUs possible. Created by the Company.

fuzzy image comparison Image comparison with the goal to check whether two images are to some extent similar to each other (but not necessarily the same).

GUI Graphical User Interface. Presents information to the user of a system by displaying text and images on a screen. Can be interacted with by using the touch screen.

HMI Human Machine Interface. A system that serves as an interface between a human operator and a machine. May take input in the form of touch-screen and hardware control events as well as actions issued by FUs. May produce visual output through a GUI and stimulate FUs through events, indications and data output.

I/O interface Input-output interface. A specification of how a certain device, system or module can be communicated with.

IDE Integrated development environment. A user interface intended to help its user write a program's source code.

indication Part of the HMI-FU data interface. A way for the HMI to share data with FUs. The format of the data is boolean. Can be compared to a function call that is meant to carry data.

jython A Java implementation of the python scripting language.

NF New Framework. The testing framework that has been developed in course of this master's thesis.

Nose A python unittest test runner. It is highly configurable when it comes to how to run tests and what kind of output to produce and allows usage of plugins to further extend its functionality.

NumPy A python library that provides linear-algebraic functions.

OpenCV Open Source Computer Vision. A library that provides computer vision functions.

oracle In a test case, a reference that a value produced by the SUT upon stimulation is compared to in order to evaluate the outcome of the test case.

PIL Python Imaging Library. A python library that provides functions for handling images.

Sikuli An automation tool able to interact with a SUT's GUI. Features an interpreter that can execute Siculi Script code, an extension of the jython language.

SUT System Under Test. A particular software or hardware system that is being tested.

TC Test Case. See: test case.

test In the context of unittest: The smallest unit of testing. Tests one single aspect of the SUT. Consists of zero or multiple stimuli and exactly one verification / assertion.

test bench The environment in which a SUT is tested. Consists of a running SUT along its test fixture.

test case In the context of unittest: A class containing one or more tests.

test fixture The controlled environment surrounding a SUT. Consists of the SUTs peripherals and any tools that may be needed for testing it.

test suite In the context of unittest: A set of test cases or other test suites.

the Company The company where this thesis was carried out.

the Editor Software produced by the Company that can be used to create HMIs.

the Engine Software produced by the Company that can be used to run HMIs.

VGT Visual GUI Testing. An automated testing technology where tests use computer vision to provide mouse and keyboard events to the SUT and verify its behaviour by analysing its visual output.

Contents

Glossary	7
1 Introduction	1
1.1 Purpose	2
1.2 Scope	2
2 Background	3
2.1 Automated testing	3
2.1.1 GUI testing	4
2.1.2 Embedded systems	5
2.2 Embedded systems at the Company	6
3 Methodology	9
4 Prestudy	11
4.1 Phase methodology	11
4.1.1 Overview of the case study project	12
4.1.2 Case study questions	12
4.1.3 Field procedures	12
4.1.4 Guide for the case study report	12
4.2 Results	12
4.2.1 Interview with the company's employees	13
4.2.2 Sikuli	14
5 Development of a testing framework	20
5.1 Key features	20
5.2 Design	23

6 Experiments	28
6.1 Phase methodology	28
6.2 Results	31
6.2.1 TC development	31
6.2.2 TC maintenance	35
6.2.3 Qualitative data	39
7 Synthesis	47
7.1 VGT of embedded systems	47
7.2 Development cost of VGT TCs for embedded systems	48
7.3 Maintenance cost of VGT TCs for embedded systems	48
8 Validity Threats	50
8.1 Internal validity	50
8.2 External validity	51
8.3 Construct validity	52
8.4 Conclusion validity	53
9 Conclusion	54
9.1 RQ1: VGT of embedded systems	54
9.2 RQ2: TC development costs	55
9.3 RQ3: TC maintenance costs	55
9.4 Future research suggestions	56
A Interview questions	58
B TC template	59
Bibliography	64

1

Introduction

The goal of software testing is to assure software quality [42, pp. 35] [11, pp. 27] and reveal defects [11, pp. 27]. However, it takes time and money to create and maintain tests and software quality is often traded-off with projects' test phases cut short [21]. This indicates a potential for improvement, where every technique that would make testing more efficient implies savings or improvements to software quality.

Numerous techniques and tools for test automation have emerged. Unit testing helps verify that low level components of a software system conform to their specifications [38] and integration testing helps verify that components function together [34]. Testing that involves graphical user interfaces (GUIs) has been more troublesome [32], where the challenges are: to locate buttons on the screen, respect the latency between a user action and the GUI's response, perceive changes on the screen and to compare them to the expected outcome. One proposed technique to match this challenge is Visual GUI Testing (VGT) [1]. In VGT, images of GUI components are stored as part of the test case (TC). A VGT testing tool uses such images and image recognition to drive the testing process and judge the test outcome. The tool can typically click elements in the SUT's GUI and give keyboard input and it can evaluate assertions about the contents of the SUT's GUI. A common problem with this approach is that VGT TC development and maintenance is costly.

Embedded systems are application-specific, dedicated to a specific task and run in a specific environment [29]. When it comes to testing of embedded systems, it is crucial to consider their non-standard input/output interfaces [29]. Testing of embedded systems may require access to the SUT by means of aforementioned interfaces, which can be difficult for generic testing tools available on the market. On the other hand, VGT can increase test coverage of automated testing, since it opens up for automated testing through the visual interface. To the author's best knowledge, there is no support for the applicability of VGT for testing of embedded systems in the current academic body of knowledge.

1.1 Purpose

The purpose of this study was to evaluate the applicability of VGT in an embedded software development context. The study was conducted with focus on: the usability of the solution, ease of TC development and maintenance. These were the study's research questions:

- RQ1: To what degree is VGT applicable for embedded systems in industrial practice?
- RQ2: What are the development costs associated with VGT TCs for embedded system environments in industrial practice?
- RQ3: What are the maintenance costs associated with VGT TCs for embedded system environments in industrial practice?

This report presents this study's methodology, background, each of its phases, a synthesis of the results of the phases, an analysis of the validity threats and finally the conclusions regarding the research questions.

1.2 Scope

Not all embedded systems feature a GUI. Since VGT is a technology that interacts with the SUT's GUI, this thesis focuses on embedded systems that have a GUI.

2

Background

This chapter describes the background of this study. The first section reports on previous work and existing literature on the topics of automated testing, GUI testing and embedded systems. The second section gives a background about the company where this thesis has been carried out.

2.1 Automated testing

The concept of automated testing has been covered broadly in literature. Meyer et al. distinguish seven items that can be automated [33]:

1. Test management and execution
2. Failure recovery
3. Regression testing
4. Script-driven GUI testing
5. Test case minimization
6. Test case generation
7. Test oracle generation.

Meyer et al. propose a testing framework that automates steps 1-3 as well as 5-7, but not step 4 [33]. The authors point out that most literature on the topic considers steps 1-3 as automated testing. They also observe that in order to enable for future regression testing of a SUT, it is important for the framework to keep record of test scenarios that produce failures.

Boyapati et al. present a framework for automated testing of Java programs [9]. The authors focus on automated TC generation and execution based on formal specifications.

The authors show that generating TCs from Java predicates is feasible even when the number of TCs possible to generate is large.

2.1.1 GUI testing

Even though most automated testing tools described in literature seem to drive the testing process programatically, the concept of automated GUI testing is not new to academia. Alégroth divides automated GUI testing into three generations [2]:

- First generation: Coordinate-based GUI testing [22]
- Second generation: Widget-based GUI testing [41]
- Third generation: Visual GUI Testing (VGT). [1]

In coordinate-based techniques (first generation) test scripts consist of a user's recorded interaction with the SUT such as keystrokes and mouse clicks. The user's actions can then be replayed, which emulates user interaction with the SUT.

The second generation approach to GUI testing as described by Sjösten-Andersson and Pareto is to have the test scripts interact directly with the code of the GUI elements of the SUT [41]. Callback functions are called in order to stimulate the GUI and check its state.

The latest generation according to Alégroth is visual GUI testing (VGT) [1], where the contents of the screen is analysed with image recognition technology. Key elements of the GUI, such as buttons and text fields, are localised and the testing tool interacts with these in order to drive the application. The state of the GUI is then compared to the expected output with image comparison algorithms.

Two examples of VGT tools are eggPlant Functional [18] and JAutomate [27]. Both tools feature their own scripting languages in which the tester can define his/her test cases. Both tools are proprietary software. At least one study evaluated the applicability of eggPlant Functional as a potential candidate to transition manual TCs to in an industrial context [3]. The study found the tool to be mature and suitable for the purpose. It also found, however, that the tool is expensive and that its scripting language has a steep learning curve and that it does not offer the modularity required for the testing purposes. Alégroth et al. present a study where they compare JAutomate to an open-source VGT tool Sikuli [13] and a commercial tool whose name the authors did not disclose [5]. The study found that JAutomate has the following advantages:

- it is platform independent
- it has record functionality
- it offers manual and semi-automated test step execution
- it has multiple image recognition algorithms
- it has images within the scripts

- it offers comprehensive failure mitigation
- it has test suite support
- it is backward-compatible.

One drawback that the study identified with JAutomate is that it does not offer support for remote connections to the SUT.

As for Sikuli, at least two experiments have been conducted that tested Sikuli for VGT in an industrial context [3, 8]. Both experiments indicate that VGT is a suitable candidate for automated GUI testing in an industrial context, but they also both report the need of further research on maintenance costs of automated test suites. Alégroth et al. also report on the high effort needed for translation of manual TCs to VGT TCs and also suggest that a technique to verify the correctness of VGT TCs better than manual verification may be required [3]. This thesis provides valuable input to the above topics in form of:

- guidelines for a TC maintenance process, in which part is verification of TC correctness
- measurements of VGT TC maintenance costs
- measurements of VGT TC development costs.

2.1.2 Embedded systems

Karmore and Mabajan list four characteristics in which embedded systems differ from general-purpose systems: embedded, interaction, development practice and timeliness [29]. According to the authors, embedded means that the system is application-specific, dedicated to a specific task and runs in a specific environment. Interaction is about communication between tasks run within the system and about their access and priority to the shared resources. Development practice can differ for embedded systems because of their specific interfaces and requirements on specific supporting platforms and tools for development. Finally, timeliness are time constraints that embedded systems may have to meet.

Emerson argues that embedded systems differ from general-purpose systems in that their hardware must often operate under special constraints regarding: limited architecture, limited cost, power consumption, processor speed and memory size [19].

Qian and Zheng propose a process for testing of embedded systems [36]. They point out that it is important to test throughout the whole development process and they propose test methodologies for the: requirement specification, design and coding phases. They distinguish host-based testing and target-based testing and encourage usage of both based on the stage of the development process. For instance, the authors suggest static code analysis as one way to host-based-test code and modeling as a way to host-based-test software design. They also suggest unit testing as a way to host-based-test code.

Emerson presents 7 tools for embedded systems that allow the tester to design, simulate, analyse and verify embedded systems [19]. The presented tools are: Generic Modeling Environment [15], Giotto [24], HyTech [23], Metropolis [7], Mocha [4], POLIS [6], and Ptolemy II [10]. These tools typically aid host-based embedded systems testing, using Qian and Zheng’s nomenclature.

As for target-based testing, Kovacevic et al. present a framework for automatic testing of set-top boxes [30]. The framework can control: power switch devices, stream modulators and a remote control emulator connected to the SUT, which is how it stimulates the SUT. In order to evaluate test outcome, the framework captures the SUT’s video and sound output and analyses their quality. It also recognises text presented by the SUT and can compare pictures output by the SUT. Karmore and Mabajan present two scenarios of target-based embedded system testing [29]. In both cases the SUT presented an interface towards a host system from which the test was driven. In the first scenario the SUT was stimulated manually through a debugging tool. In the second scenario the tool used by the authors could run the test automatically. Chakrabarty and Cheng present built-in self-tests in embedded systems [12, 14]. Legourski et al. present a testing system that could verify embedded systems created by students in an embedded systems course [31]. The tool can be connected to the target system and verify that it behaves as specified. The tool features a language in which the expected behaviour of the target system could be specified. Zainzinger presents a similar tool able to interpret test scripts written in C and execute them on a target system, provided that the target system implements an interface to the host system from which the testing process is driven [45].

Despite the effort put in examining the existing literature on the topic, the author of this report has been unable to find any studies that evaluate the applicability of VGT to testing of embedded systems.

2.2 Embedded systems at the Company

This thesis has been carried out at a Gothenburg-based company that operates in the automotive industry (referred to as ”the Company”). The Company produces software for development and running Human Machine Interfaces (HMIs) for cars. An HMI in this context is a system that supports its user’s interaction with his/her car and its subsystems.

As shown in Figure 2.1, a typical HMI provides the following I/O interface:

- visual (visual output and touch screen event input)
- hardware controls (physical buttons and rotary controls)
- functional units (FUs) – a generic interface to a car’s subsystems.

Figure 2.2 shows how HMIs are typically deployed in a car. An HMI is a specification of possible I/O, GUI transitions and logic in the system and is run by a program called the Engine. The Engine takes input, processes data and renders the GUI and other

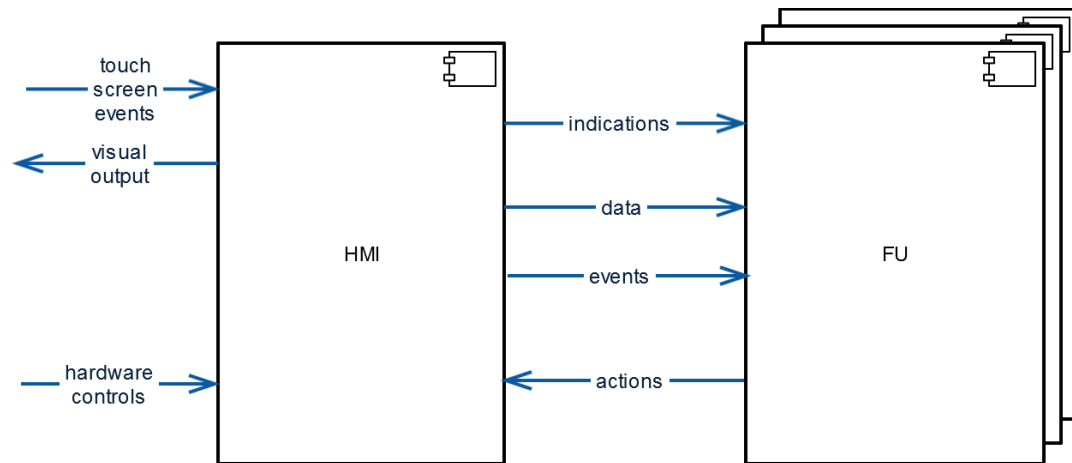


Figure 2.1: An HMI's possible I/O interface

output according to the HMI's specification. The Company has developed the Editor – an integrated development environment (IDE) for HMIs – and "fusdk" – a C++ and Python library that is the layer of communication of a car's physical subsystems with the HMI.

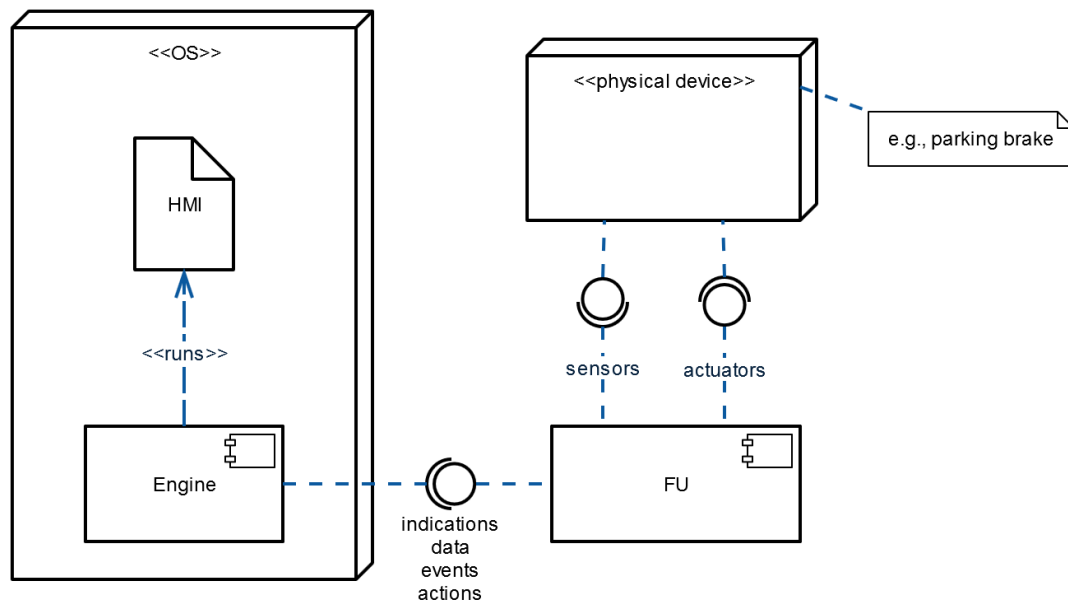


Figure 2.2: A typical deployment of an HMI in a car

The Company tests HMIs manually, where the tester stimulates the HMI and verifies the output. The Company also uses an in-house-developed automated testing framework,

where the stimulation and verification are defined in a test script that is executed by the framework. The testing framework is based on Robot Framework [37]. It can stimulate the SUT through the hardware-controls and FU interfaces and can capture screenshots of the SUT's GUI and do exact image comparisons to oracle images, but it cannot provide the SUT with input through the visual interface, so the framework does not support VGT.

The Company desires the following features in a testing framework, based on the problems recognised with their current testing practices:

- It is necessary to test the SUTs through the visual interface, since this interface is important in interaction with the end-user. Visual testing is currently done manually, which the employees judge as error-prone, because it happens that testers miss errors presented by the SUT's GUI that are important to correct before the product is released to a customer. Hence, there is a demand at the Company for a tool that would enable automated visual testing of embedded systems.
- It should be possible to test other interfaces than solely the visual.
- The employees found it difficult to get started with the current testing framework and also desired a way to develop new TCs quickly, so there is potential for making the testing process easier and faster.
- It is desirable for tests to be easy to reuse and maintain.
- Informative reports should be produced from test runs.

3

Methodology

This section presents the methodology of the study. As pictured in Figure 3.1, this was a multi-phase study with a prestudy, development of a testing framework and experiments that compared two frameworks. The Company is seen as an instance of industrial practice with respect to this study's research questions. The prestudy was done in order to acquire context information about VGT's applicability in the context of embedded systems at the Company. This information was then used to develop a new VGT framework. The new framework was then compared to Sikuli in the experiments. During the experiments TC development and maintenance cost metrics were collected as well as qualitative observations noted. The information from all the study phases was then analysed, synthesised and conclusions drawn and presented in this report.

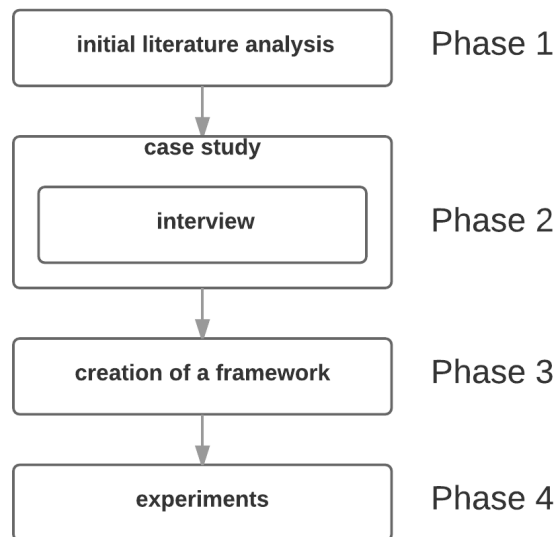


Figure 3.1: The steps carried out within this study.

Each of the phases of this study as pictured in Figure 3.1 was conducted by two students with a similar educational background of nearly completed five-years graduate studies in computer science. According to Höst et al., under certain conditions, software engineering students may be used in empirical studies instead of professional software developers [26]. One of the subjects had more working experience than the other; about seven years as compared to the other subject's one year.

4

Prestudy

This chapter describes the phase methodology and results of the prestudy that was carried in this thesis.

4.1 Phase methodology

A prestudy has been conducted to acquire context information about VGT's applicability in the context of embedded systems at the Company. This section explains how this prestudy was conducted.

The prestudy was designed as a case study with Yin's framework for case study design [44, chapter 2], which requires the definition of:

- The study's questions,
- its propositions, if any, or otherwise its purpose in case of exploratory case studies,
- its unit(s) of analysis,
- the logic that links the data to the propositions, and
- the criteria for how the findings will be interpreted.

Since the case study was exploratory in its nature, its purpose along with success criteria were stated instead of propositions.

The case study is presented in the form of a case study protocol as described by Yin [44, pp. 79-90], which consists of:

- overview of the case study project,
- field procedures,
- case study questions and

- guide for the case study report.

The reminder of this section follows this outline.

4.1.1 Overview of the case study project

This case study was conducted in order to investigate the applicability of a VGT tool for testing of embedded systems at the Company. Because of the resource constraints of this study, Sikuli was investigated as it was the only open-source VGT tool that this study has identified. The cases evaluated were two HMIs available at the Company and the unit of analysis was the applicability of Sikuli to testing of those systems.

4.1.2 Case study questions

The case study question was:

- What technical and process related practices should be used to apply VGT in the context of embedded systems in industrial practice?

4.1.3 Field procedures

An interview was conducted with two HMI developers in order to find out about:

- the testing tools and practices at the Company
- VGT's applicability to testing of the Company's embedded systems
- the expectations on a tool for testing of the Company's HMIs.

The questions asked during the interview can be found in Appendix A. The interview was conducted by the subjects described in section 3.

The functionality offered by Sikuli [40], especially in the context of testing of HMIs, was investigated during half a working day. Two simple TCs were implemented for one of the company's HMIs. The TCs were implemented by the subjects described in section 3.

4.1.4 Guide for the case study report

Any observations related to the research questions were noted down and the notes analysed using document analysis [20]. The results of this analysis are reported in section 4.2.

4.2 Results

This section presents the results of the prestudy. First, the results of the interview are presented. Then the findings about Sikuli and its applicability for testing of embedded systems at the Company are presented. Table 4.1 in the end of this section summarises the results of the prestudy.

4.2.1 Interview with the company's employees

This section presents the results of the interview conducted with the Company's employees.

Testing at the company

The Company applies manual testing in connection to HMI development, which is considered good for testing of functional parts of the SUTs, but unsatisfying for the visual aspects of SUTs, as it is difficult for the naked eye to discover deviations from the expected graphical output when the differences are small. The Engine renders GUIs differently each time and the Company wants to be able to control that variation. VGT may be a good alternative to achieve that.

In one project real hardware FUs were stimulated that in turn interacted with the HMI in order to tested it. This however meant that a higher level of abstraction of the SUT was used, hence abstracting away the important details of the HMI-FU communication. This also introduced an additional source of errors into the testing, because errors could also occur due to errors in the FU implementation.

In another project a scriptable FU Simulator was used, where the HMI could be stimulated through the FU interface from a script. The setup sped up testing, but it only supported the FU interface to the HMI and required much human interaction to drive the tests and verify their outcome. It was perceived convenient, however, to be able to encapsulate multiple steps of FU-HMI communication in callable functions. On the other hand, it was noted that as HMIs increased in complexity, it became difficult and time-consuming to write and maintain FU-simulator scripts.

Old testing framework The existing testing framework has been used in one project, but the interviewees have not used it because of the steep learning curve. Moreover, it was found inconvenient that the framework used two languages (Python and Robot-Framework's own). Finally, it was perceived as a drawback that the existing testing framework could not drive tests through the visual interface. Instead, it could drive tests through the FU and hardware control interfaces and do assertions on output from the FU interface. It could capture images, but they could only be compared manually with an external program.

Expectations on a testing framework

Regression testing was the main expected usage for an automated testing framework, with mainly small TCs that should be possible to run quickly. TC development should be as simple and quick as possible. The testing framework should support the whole HMI I/O interface (see Figure 2.1). Two components of HMI testing were listed; the functional and the visual and it was stressed that both components should be tested.

The following features were perceived desirable in a testing framework:

- the ability to verify GUI transitions and animations

- arbitrary-shape masks that would make it possible to test one part of the screen at a time
- the ability to extract screenshots from test outcomes for communication with stakeholders.

4.2.2 Sikuli

Sikuli uses visual technology to automate and test GUIs using screenshot images [40]. It features an IDE and interpreter of Sikuli Script – a Jython-based scripting language. The language is extended with functions for interaction with GUIs such as:

- `click()` - find and click an image
- `dragDrop()` - perform a drag-and-drop between the two images
- `find()` - find an image
- `type()` - type text.

Sikuli provides a feature that helps capture reference images. When activated, Sikuli is minimised and waiting for the user to mark the area that should become the new reference image as shown in Figure 4.1.

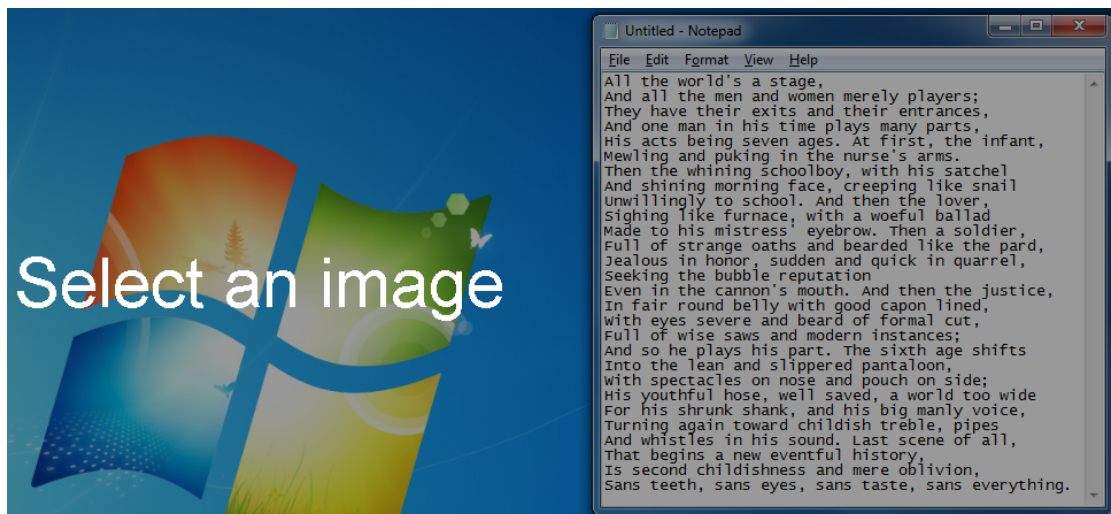
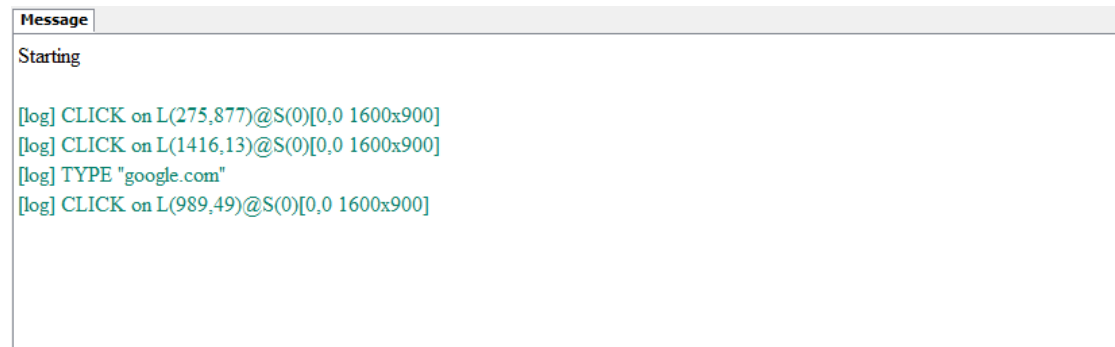


Figure 4.1: Sikuli in a minimised state waiting for the user to mark the area that should be used as a reference image.

Sikuli does not provide any way to structure test scripts. In order to enable for modularisation of test scripts, Jython's native unittest library was applied for the purpose of this study. Likewise, Sikuli's reporting capabilities were judged poor (e.g., Figure 4.2). In order to increase its usability for testing, HTMLTestRunner – an extension library to unittest that provides simple html reporting [25] – was used for the purpose of this study.

Figure 4.3 presents an example Sikuli test run report generated with HTMLTestRunner. Figure 4.4 presents a typical test bench set up for Sikuli as used in this study. Since the tool does not support the Python library ctypes that fusk depends upon, it could not be used directly to set up the test fixture. Therefore in this study a system call was used to invoke a Python script that did the setup.



```

Message
Starting

[log] CLICK on L(275,877)@S(0)[0,0 1600x900]
[log] CLICK on L(1416,13)@S(0)[0,0 1600x900]
[log] TYPE "google.com"
[log] CLICK on L(989,49)@S(0)[0,0 1600x900]

```

Figure 4.2: An example log of a successful Sikuli script run.

When it comes to supported interfaces, Sikuli can analyse the screen content, control the mouse and keyboard and execute Python code. Since it cannot use fusk, Sikuli cannot access the FU, nor the hardware controls interfaces on a code basis. For access to the hardware controls interface, however, a built-in feature called faceplate was activated in the SUT. This feature adds a component to the HMI's GUI that simulates hardware controls. Sikuli was able to interact with that GUI component and hence simulate hardware control events to the SUT. It was judged important to be able to test the FU-HMI interface for a good test coverage, but for meaningful testing it was enough to access the hardware controls interface and the visual interface.

Conclusion	Description	Interview	Sikuli tests
CS ₁	Requirement: Manual testing works well for functional parts of the SUTs, but is insufficient for the visual interface due to the limitations of human vision. The Company wants to control the image rendering variations. VGT may be a good alternative to achieve that.	X	X
CS ₂	Requirement: The FU-HMI interface must be tested. Using real FUs obscures the communication on that interface and introduces a new source of errors.	X	X

CS₃	To be able to script the behaviour of FUs helped raise the level of abstraction of the HMI-FU communication, especially when setting up the HMI to its initial state.	X	
CS₄	Development and maintenance of FU Simulator scripts becomes difficult with increasing complexity of the SUT.	X	
CS₅	The steep learning curve of a testing framework hampered the desire to learn how to use it.	X	
CS₆	Requirement: One scripting language should be used.	X	
CS₇	Requirement: The framework should help create and run small test cases quickly.	X	
CS₈	Requirement: The framework should support the full I/O interface of the HMIs.	X	X
CS₉	Requirement: The framework should support comparing custom areas of the GUI.	X	X
CS₁₀	Requirement: The framework should be able to export screenshots from test outcomes.	X	
CS₁₁	For the Company's SUTs meaningful tests can be created that only test the hardware controls interface and the visual interface. For a satisfactory level of test coverage, however, the FU interface should also be tested.	X	X
CS₁₂	Sikuli can analyse the content of the screen, simulate mouse and keyboard events and execute Jython code. Sikuli cannot natively access the SUTs' hardware controls interface, but this interface can be simulated by items added to the HMI's GUI, which Sikuli can access.		X
CS₁₃	Sikuli lacks a way of grouping of test cases and features poor reporting on test run results. This result is tool-dependent and has been shown for Sikuli.		X

CS₁₄	Sikuli does not support communication with FUs, nor input through hardware controls. This result is tool-dependent and has been shown for Sikuli.		X
CS₁₅	Reference images in Sikuli are obtained in SUT run-time through region captures from Sikuli.		X

Table 4.1: A summary of the results of the prestudy. The results with an 'X' in the Interview column stem from the interview and the results with an 'X' in the Sikuli tests column stem from the Sikuli tests. The results with the **Requirement** label are requirements expressed by the interviewees.

MyTitle

Start Time: 2014-08-19 00:53:02

Duration: 0:07:58.373001

Status: Pass 20

This is a description.

Show [Summary](#) [Failed](#) [All](#)

Test Group/Test case	Count	Pass	Fail	Error	View
Experiments	20	20	0	0	Detail
test_sa			pass		
test_sb			pass		
test_sc			pass		
test_sf: Clicking on the third element in the Reset menu marks it.			pass		
test_si: Short pressing B closes the Reset menu.			pass		
test_sk: Scrolling down twice shows TC3			pass		
test_ta: Navigation to Oil level menu			pass		
test_tc: While in the Service status menu, short-pressing A exists the menu			pass		
test_td: Long-pressing B (more than 1s) brings up the Trip computer reset menu			pass		
test_te: Clicking on the second element in the Reset menu marks it.			pass		
test_tf: Clicking on the second element in the Reset menu marks it.			pass		
test_tj: Scrolling down once shows TC2			pass		
test_ub: Message menu option exists			pass		
test_uf: Parking heater menu option exists			pass		
test_ug: Parking heater menu option can be entered			pass		
test_uh: Theme selection highlight Eco			pass		
test_uk: Menu has circular navigation			pass		
test_vd: The Performance theme in the Themes menu is clickable			pass		
test_vg: Parking heater menu option can be entered			pass		
test_vk: Menu has circular navigation			pass		
Total	20	20	0	0	

Figure 4.3: An example Sikuli test run report generated with HTMLTestRunner.

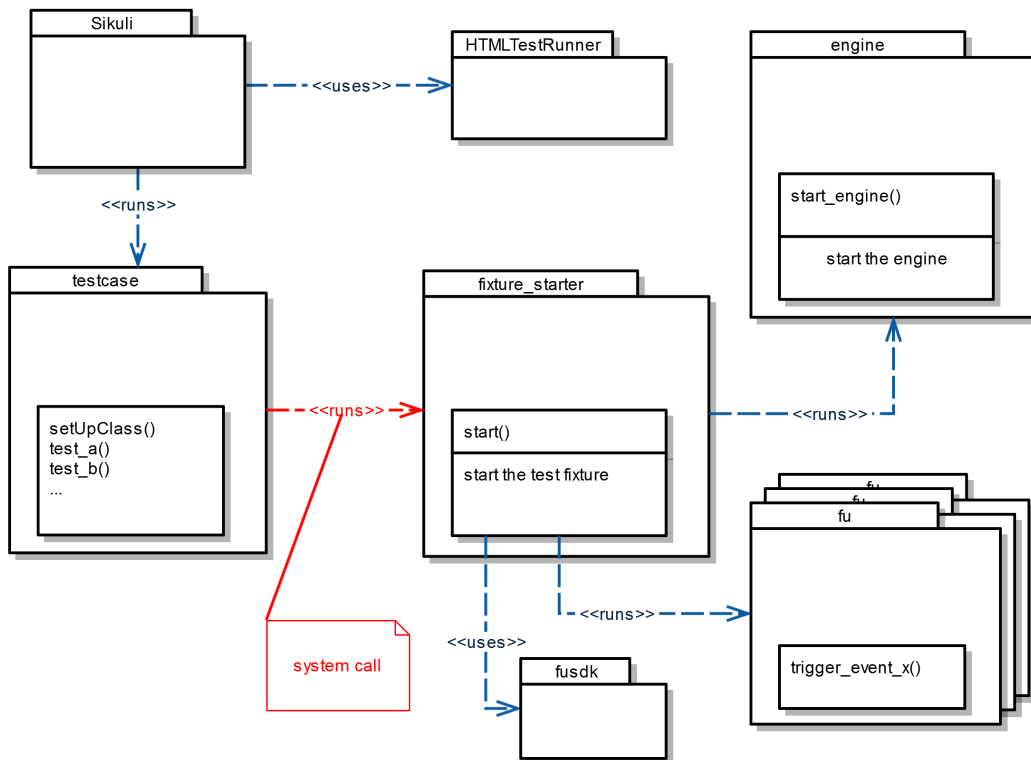


Figure 4.4: A typical test bench set up as used in this study; with Sikuli, unittest and the HTMLTestRunner.

5

Development of a testing framework

Since Sikuli was not able to communicate with the SUT through the FU interface (CS_{14}), which is an important interface to test (CS_{11}), a new testing framework was developed. This section presents the resulting framework. The text is divided into four parts: the framework's key features, design, implemented TC development process and TC maintenance process. Table 5.2 at the end of this section presents its summary.

5.1 Key features

Table 5.1 presents the key features implemented in NF. Each of the feature is then discussed in detail in the following sections.

Feature₁	A process for TC development
Feature₂	A way to structure TCs
Feature₃	Support for full HMI I/O interface
Feature₄	Support for external image comparators
Feature₅	Test run reports
Feature₆	A process for TC maintenance

Table 5.1: The key features implemented in NF.

A process for TC development

In order to help the tester get started writing TCs, NF can generate a test fixture and an example TC for a given SUT. In order to generate these artefacts, NF needs:

- the location of the Engine
- the location of the HMI
- a description of the SUT's I/O interface.

The first item is the path to where the user has installed the Engine. The last two items are typically output from the Editor in form of the file that the Engine should run and a set of files that contain headers for functions that can be called by the SUT and the FUs. Provided with the required items, NF can generate the following files:

- engine.py - controls the Engine
- fixture.py - controls the test fixture – the Engine along with its peripherals (FUs)
- start_hmi.py - starts the Engine, HMI and its peripherals
- runner.py - runs test suites
- extest.py - ready-to-run example test suite

The generated example TC contains examples of SUT stimuli and assertions and is runnable as is, so one way to get started with TC development is to copy-paste and modify the example TC's code.

A way to structure TCs

NF can run Python's unittest TCs. The unittest library therefore provides a way for the tester to structure his/her NF TCs. Moreover, example TCs generated by NF follow the unittest standard in order to provide structure among TCs.

Support for full HMI I/O interface

The tester can write NF TCs that reach the full HMI I/O interface (presented in Figure 2.1). For the visual interface the framework supports VGT. It can match images either exactly or do "fuzzy" comparisons with customisable tolerance.

Support for external image comparators

In order to make the testing framework sensitive to the exact type of image differences, NF supports usage of external image comparators as long as they conform to the provided API.

Test Report

Start Time: 2015-10-06 18:51:18
Finish Time: 2015-10-06 18:51:25

Test Group/Test case	Time	Count	Pass	Fail
[-] all_tests_test_suite:BasicFunctionsTest	6.08	7	7	0
[+] setUpClass: This method is called before any test method in the class is run.	2.01	PASS		
[+] test_call_monitoring: Capture a simple action.	4.00	PASS		
[+] test_similar_text_after_changes_is_displayed: Assert text is displayed after changes - 0.9 similarity	0.01	PASS		
[+] test_sub_image:	0.00	PASS		
[+] test_text_after_changes_is_displayed: Assert text is displayed after changes	0.01	PASS		
[+] test_text_is_displayed: Assert text is displayed	0.01	PASS		
[+] tearDownClass: This method is called after all the test methods in the class have been run.	0.02	PASS		
Total	6.08	7	7	0

Figure 5.1: A typical report produced by NF.

Test run reports

In order to provide the tester with information about the failing TCs, the framework produces test run reports as exemplified in Figure 5.1. The report contains expandable sections, one for each test case. Each section presents the stdout, any exceptions raised during the execution and the image comparisons as shown in Figure 5.2. For each comparison the expected, actual and difference images are presented, as well as the similarity. Actual image is the part of the SUT's GUI that NF judged as most similar to the expected image. Any missing images will be reported as shown in Figure 5.3. The reports are self-contained HTML pages, i.e. they only need a browser to be displayed.

A process for TC maintenance

NF features a TC maintenance process that is meant to be as fast as possible. The maintenance workflow depends on why the test is failing. For increased visibility the framework raises an exception for each TC that fails and presents it in the report along with a stacktrace. Such exceptions need to be traced and resolved.

For missing reference images (e.g., Figure 5.3) the framework reports a special type of exception, saves the actual image and displays it in the report. If the actual image looks correct, the tester should copy-paste it to the reference image directory for future test runs. This step was kept manual in order to keep the test run reports dependent only on a web browser in order to assure portability of the reports. If this step was to be automated, a program would need to be running that could copy-paste the new reference image automatically.

```

❑ choose_biometric_devices: 0.00 FAIL

❑ Test output:
21:33:10,125: INFO: testfixture.Screenshot().get_screenshot() now...
21:33:10,194: INFO: 440
21:33:10,194: INFO: 52
21:33:10,194: INFO: FU_STATES:
21:33:10,214: ERROR: Image comparison 0
Traceback (most recent call last):
  File "C:\Python27\lib\unittest\case.py", line 329, in run
    testMethod()
  File "C:\Python27\lib\site-packages\populustestingframework\testcase.py", line 30, in inner
    method(self, method.__name__)
  File "C:\Users\maciej.makarewicz\Documents\prv\mt_resources\git_repos\HMI Testing Framework
\test_suites\prf\desktop\all_tests_test_suite.py", line 159, in choose_biometric_devices
    self.assertSimilar(actual_small, ref, Comparator(0.99))
21:33:10,214: DEBUG: File "C:\Python27\lib\site-packages\populustestingframework\testcase.py", line 67, in inner
    method(self, *args, **kwargs)
  File "C:\Python27\lib\site-packages\populustestingframework\testcase.py", line 49, in inner
    method(self, comparison_number=new_id, *args, **kwargs)
  File "C:\Python27\lib\site-packages\populustestingframework
\testcase.py", line 162, in assertSimilar
    raise ScreenAssertionError(comparison_number)
21:33:10,214: ERROR: ScreenAssertionError: Image comparison 0 failed.

❑ Image comparison 0:
[failed] because the screen looked significantly different than expected. Similarity: 0.950986684899

Expected: Difference: Actual:


```

Figure 5.2: The details about a failed test as presented by NF.

For test runs where the framework reports a significant image difference there are several solutions. If the difference was unexpected, maintenance is required for the SUT. It could also be that the image comparison algorithms were too sensitive to differences that are of no interest for the test, in which case the tester can use fuzzy instead of exact image comparison or tune the comparator’s sensitivity if fuzzy image comparison is already being used. Another option is to implement and use an own image comparison algorithm.

Finally, a test may fail because of an outdated reference image after a SUT update. In that case, the maintenance required is similar to that of a missing reference image; the actual image needs to be copied from the last_run directory to the reference image directory.

5.2 Design

Figure 5.4 presents an overview of NF’s internal subsystems and its usage of 3rd party software. Generator is a module that has been built for the purpose of this study that uses its templates and information about the SUT provided by the tester to generate a test bench as explained in section 5.1. The test bench files are generated with python’s string.Template functionality [35] based on the following template files:

- a template file with functions to manage the Engine (start, stop)

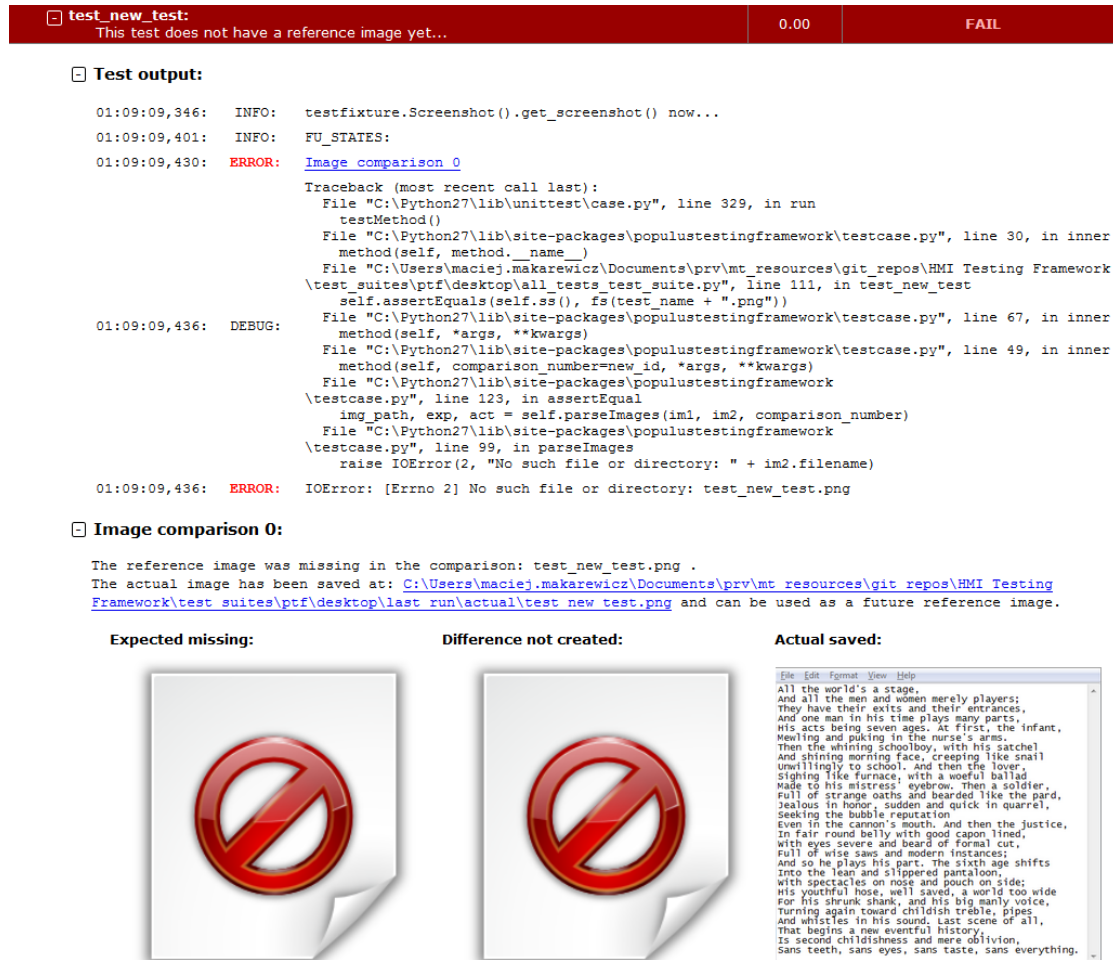


Figure 5.3: A missing reference image as reported by NF.

- a template file with functions to control the test fixture (start, stop, set up)
- a template file with functions to start the HMI, Engine and its peripherals
- a template file with a function to start a unittest TC
- a template file that defines a typical unittest TC.

Figure 5.5 shows the design of a generated test bench. The core of NF is unittest – Python’s native unit test library. NF extends this library with better logging support and VGT functionality. NF tests are run by Nose – an open-source unittest test runner. A Nose plugin, nose_html, has been developed for extended reporting capabilities. Python Imaging Library, PIL, is used for managing images. Numpy is used for linear-algebraic image comparison computations. OpenCV is used for VGT capabilities, e.g. to find

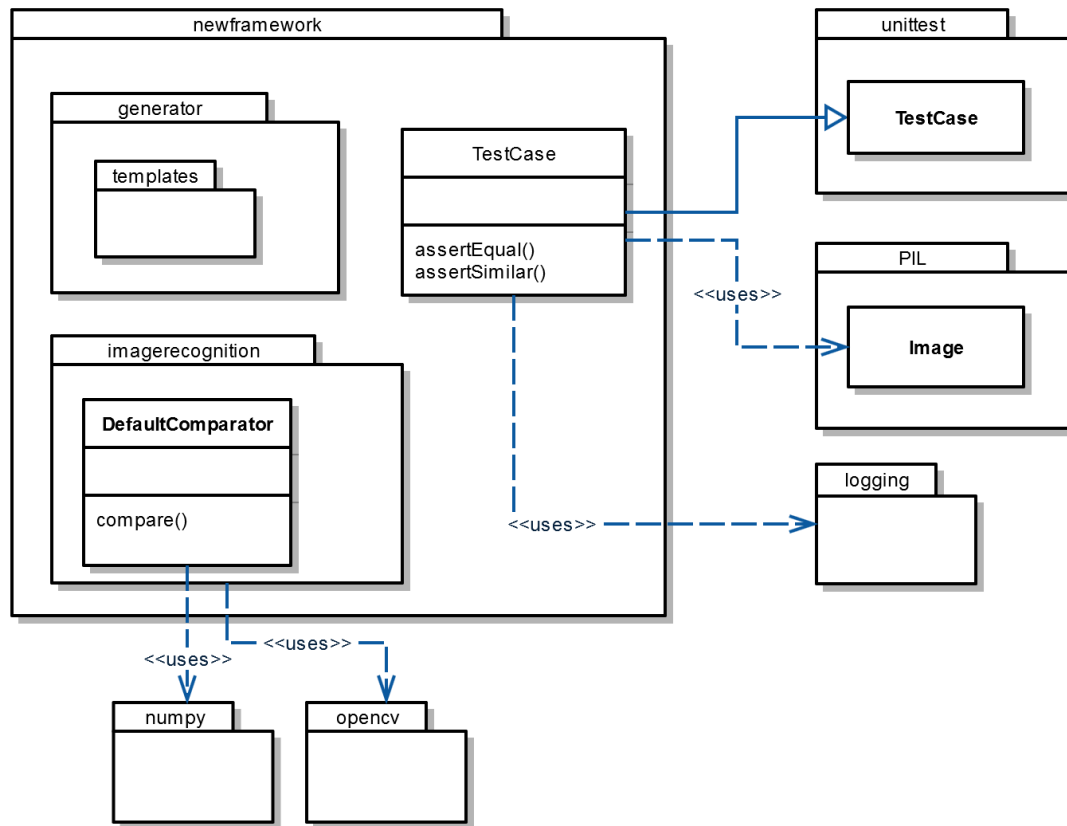


Figure 5.4: An overview of NF's subsystems and its usage of 3rd party software

images in images. It is also used in the default image comparator to find the similarity of two compared images. Finally, fusk is used for communication with FUs. This is visualised in Figure 5.4 and Figure 5.5.

Conclusion	Description
NF ₁	NF can generate an example test suite and scripts that control the test fixture and the Engine, and start the HMI and tests.
NF ₂	NF supports the full I/O interface of an HMI.
NF ₃	NF's default image comparator has adjustable sensitivity. NF also allows the usage of external image comparators.
NF ₄	NF's test run reports present the stdout, exceptions raised and image comparisons conducted during test runs. For image comparisons the expected, actual and difference images and the similarity are presented. The reports need only a web browser to be displayed.

NF₅	After each test run NF saves the actual images in a specific directory, so they can be directly used as oracles in future test runs.
NF₆	NF extends Python's library unittest with better logging support and VGT. TCs are run by the test runner Nose extended with a plugin for producing HTML test run reports.
NF₇	NF uses Python Imaging Library, PIL, to manage images and Numpy for optimised linear-algebraic image comparison computations. OpenCV is used for VGT capabilities.
NF₈	NF uses fusk for communication with FUs and the Engine.
NF₉	NF's TC maintenance process is to trace and resolve the exceptions raised during TC execution. NF raises and presents specific exceptions for missing reference images and actual images that differ significantly from expected images. For missing reference images the maintenance process is to copy-paste the images from the last_run directory to the TC directory. Significantly different actual and expected images could be a result of a bug in the SUT, in which case SUT maintenance is necessary. It could also be a result of a valid change in the SUT, in which case the new reference image needs to be copied from the last_run directory. Finally, it could be a result of too sensitive image comparison algorithms, in which case their sensitivity can be decreased or an external image comparator can be used that is more suitable for the type of image differences in context.

Table 5.2: A summary about the resulting framework.

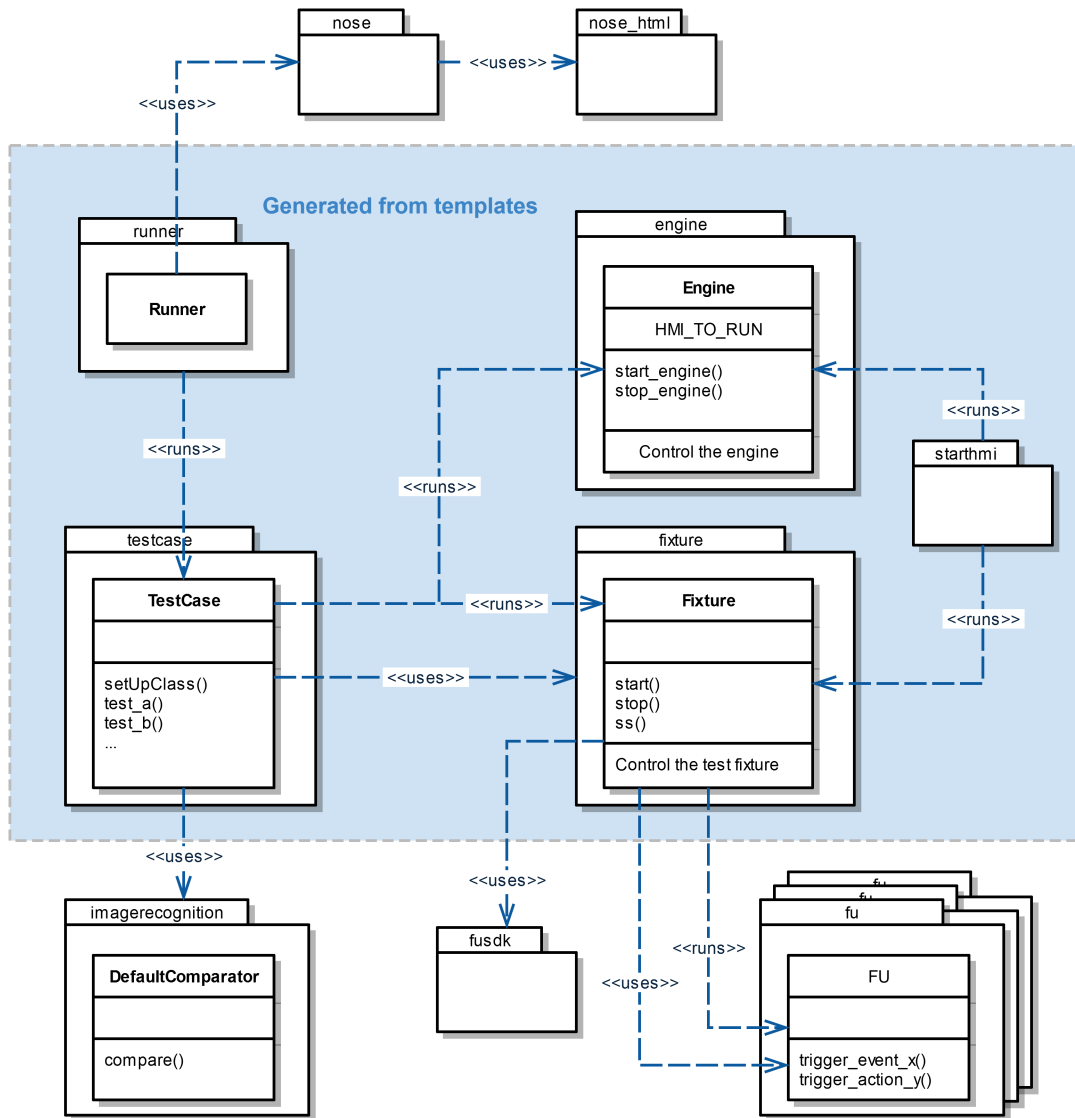


Figure 5.5: An overview of a test bench generated by NF

6

Experiments

This chapter presents the phase methodology, the results and analysis of the experiments conducted in this thesis.

6.1 Phase methodology

In order to answer RQ2 and RQ3, two experiments were set up, one for each research question, where two testing frameworks were compared:

- NF (described in section 5)
- Sikuli extended with unittest and better reporting capabilities (described in section 4.2.2).

This section describes how the experiments were conducted and their results analysed.

TC development

The TC-development experiment was to show whether the choice of the testing framework can lower the costs of TC development. A randomised paired comparison design [28, section 5.3.2] was used. Using Wohlin et al.'s nomenclature [43, pp. 74-76], the experimental objects were 40 TCs that the experimental subjects defined in natural language according to a template (Appendix B) with the goal to make them as representative as possible for typical TCs at the Company with respect to TC size and complexity and type of stimuli and assertions. The TCs were written for a commercial HMI. In order to assure the desired representativeness of the TCs, the subjects defined them based on their knowledge about the SUT and about typical TCs at the Company. The subjects acquired information about the SUT through qualitative analysis of its requirements specification and they acquired information about the Company's typical TCs through

qualitative analysis of a test suite for the SUT written for the old framework and through observations of the Company’s manual testing process. The TCs were chosen to cover all typical interfaces to an HMI (described in section 2.2) except for the FU interface which Sikuli could not connect to (CS₁₄). The defined test suite’s representativeness was validated by the Company’s employees. The dependent variable was the time it takes to implement one TC in a testing framework. The studied factor was the choice of the testing framework that the TC was to be implemented in and the applied treatments of the factor were NF and Sikuli.

The two compared frameworks had two main differentiators:

- NF’s test run reports present more information (NF₄) than Sikuli’s (CS₁₃)
- Sikuli has a different process for obtaining reference images (CS₁₅) than NF has (NF₅, NF₉).

In order to provide input to RQ2, the following null hypothesis was used:

- H_{C_0} : There is no statistically significant difference between NF’s and Sikuli’s development times of TCs for embedded systems.

Depending on the outcome of the evaluation of H_{C_0} this study would provide support whether or not the aforementioned differentiators between NF and Sikuli have an influence on VGT TC development times. Development time is seen as a metric for development cost here.

The two subjects described in section 3 were used as experimental subjects. As mentioned in section 3, one of the subjects (Subject 1) had more working experience than the other. In order to minimise the effect of this potential undesired variation, both subjects implemented the same number of TCs and applied both treatments to the same extent.

Each TC was implemented once in Sikuli and once in NF. Which TC would be implemented by which subject using which framework was randomised. The implementation times were taken from the moment the subject started to implement the TC until it was implemented according to the template and was passing when run. This design meant that each TC generated a pair of data points; one for Sikuli and one for NF.

Another approach for the TC development experiment would have been to let each subject implement each TC twice in each framework, but this was not wanted in order not to introduce the bias where the subject gets familiar with one implementation of the TC before implementing it again in the other framework.

TC maintenance

The TC-maintenance experiment was to show whether the choice of the testing framework can lower the costs of TC maintenance. A simple randomised design [28, section 5.3.1] was used. Using Wohlin et al.’s nomenclature [43, pp. 74-76], the experimental objects were TC updates necessary to carry out after migrations to newer versions of the SUT. The dependent variable measured in this experiment was the time it takes to

maintain one TC in a testing framework, measured from a failing state to a passing state. The studied factor was the choice of the testing framework to maintain the TC in and the applied treatments of the factor were NF and Sikuli. The same subjects conducted the experiment as in the case of the TC development experiment.

In order to provide input to RQ3, the following null hypothesis was used:

- H_{M_0} : There is no statistically significant difference between NF's and Sikuli's maintenance times of TCs for embedded systems.

Similar to H_{C_0} , the outcome of the evaluation of H_{M_0} would provide support for whether or not the differentiators between NF and Sikuli (listed in section 6.1) have an influence on VGT TC maintenance times. Similarly, maintenance time is seen as a metric for maintenance cost in this study.

The 80 TC implementations from the previous experiment and an initial version of the SUT were used as starting point. Next, the SUT was updated to a next version and the TCs were ran. If any of the TCs failed, a simple root-cause analysis was done. If the TC failed due to changed SUT behaviour such as:

- an element appeared or disappeared from the SUT's GUI
- the SUT entered a different state (e.g. presented a different GUI) as a result of a stimulus or stopped reacting to the stimulus
- an element in the SUT's GUI changed significantly,

the subject whose TC failed carried out maintenance on it in the framework where the TC failed. The maintenance carried out was to update the reference image or to update the TC code. The above definition of changed SUT behaviour means that maintenance moments due to small differences in how the Engine rendered the HMI were disregarded with respect to the data collection process. This choice was done in order to mimic the practice at the Company where a certain degree of variation was allowed for how the HMI was rendered.

Ideally TCs should be tracable to the requirement specification document, but since the TCs used in the experiments were only based on the SUT's requirement specification document and not an exact subset of the requirements from the requirement specification document, not necessarily all TCs could have been traced back to exact requirements. For this reason the data collection process was made error-driven rather than requirement change-driven.

Once maintenance was carried out on a TC, the SUT was updated to a next release and the process continued until 12 SUT updates have been carried out. Each time a TC was updated, it became the new baseline for the experiment, i.e. the subsequent versions of the SUT were always run on the most recent version of the TC. This choice was made in order to follow the industrial practice. Data about the type of defects found by the respective framework was recorded throughout the experiment and on a high level compared to the type of defects found by manual testing and testing with the Old Framework. This comparison was done by the experimental subjects and cross-checked with employees at the Company.

Data analysis

The two experiments generated a total of four data samples. The Shapiro-Wilk normality test [39] with a significance level $\alpha = 0.05$ was applied to each sample to check whether parametric or non-parametric tests should be used for data analysis. For each of the four samples the following null hypothesis was used:

- $H_{s,\text{normality}_0}$: The sample s comes from a normally distributed population.

As explained in section 6.1, the data obtained from the TC-development experiment came in pairs – one data point for NF and one for Sikuli. Therefore, a matched-pairs test was used for analysing the data; paired T-test [17] for data from a normally distributed population and Wilcoxon Matched-pairs Signed-ranks test [28, pp. 328-330] otherwise.

Since the data obtained from the TC-maintenance experiment did not necessarily come in pairs, matched-pairs tests could not be used in that experiment. Hence, the T-test [16] was used for data from a normally-distributed population and the Mann-Whitney U-test [28, pp. 324-326] was used otherwise. A significance level of $\alpha = 0.05$ was used for analysing the data from both experiments.

6.2 Results

This section presents the results of both experiments along with an analysis of the results. Table 6.3 in the end of this section summarises the results of the experiments.

6.2.1 TC development

Table 6.1 presents the times it took each person to develop each TC in respective tool.

Test Case	Subject 1		Subject 2	
	Framework	Time	Framework	Time
TC ₁	NF	6:30	Sikuli	15:22
TC ₂	NF	9:24	Sikuli	6:04
TC ₃	NF	5:12	Sikuli	3:36
TC ₄	Sikuli	6:06	NF	11:19
TC ₅	Sikuli	11:00	NF	11:17
TC ₆	NF	10:00	Sikuli	10:27
TC ₇	Sikuli	17:30	NF	11:20
TC ₈	NF	3:36	Sikuli	5:48
TC ₉	Sikuli	6:30	NF	2:35
TC ₁₀	NF	5:30	Sikuli	3:09
TC ₁₁	NF	8:00	Sikuli	6:04

Test Case	Subject 1		Subject 2	
	Framework	Time	Framework	Time
TC ₁₂	NF	5:45	Sikuli	1:57
TC ₁₃	NF	2:15	Sikuli	2:28
TC ₁₄	NF	5:45	Sikuli	2:22
TC ₁₅	NF	5:48	Sikuli	3:07
TC ₁₆	Sikuli	8:18	NF	6:58
TC ₁₇	Sikuli	5:24	NF	3:58
TC ₁₈	Sikuli	9:12	NF	3:32
TC ₁₉	NF	4:30	Sikuli	2:54
TC ₂₀	Sikuli	4:36	NF	3:48
TC ₂₁	NF	6:24	Sikuli	5:06
TC ₂₂	Sikuli	12:00	NF	18:36
TC ₂₃	Sikuli	10:00	NF	7:40
TC ₂₄	Sikuli	9:30	NF	6:06
TC ₂₅	NF	2:36	Sikuli	9:15
TC ₂₆	NF	3:00	Sikuli	3:23
TC ₂₇	NF	6:48	Sikuli	4:50
TC ₂₈	Sikuli	13:12	NF	5:58
TC ₂₉	Sikuli	7:24	NF	4:15
TC ₃₀	NF	8:00	Sikuli	5:42
TC ₃₁	Sikuli	5:36	NF	6:22
TC ₃₂	Sikuli	7:12	NF	3:10
TC ₃₃	NF	9:30	Sikuli	3:18
TC ₃₄	Sikuli	6:00	NF	4:00
TC ₃₅	Sikuli	6:12	NF	4:11
TC ₃₆	NF	4:00	Sikuli	5:47
TC ₃₇	Sikuli	5:00	NF	3:35
TC ₃₈	Sikuli	5:30	NF	4:20
TC ₃₉	Sikuli	4:48	NF	2:30
TC ₄₀	NF	4:12	Sikuli	2:30

Table 6.1: The results of the TC-development experiment. Subject is the subject who carried out the implementation, Framework is the framework used and Time is the time the implementation took.

Analysis

Figure 6.1 shows the boxplots of the data points for NF and Sikuli. The boxplots indicate that both samples contain one outlier each:

- TC22 for NF by Subject 2 (1116s)
- TC7 for Sikuli by Subject 1 (1050s).

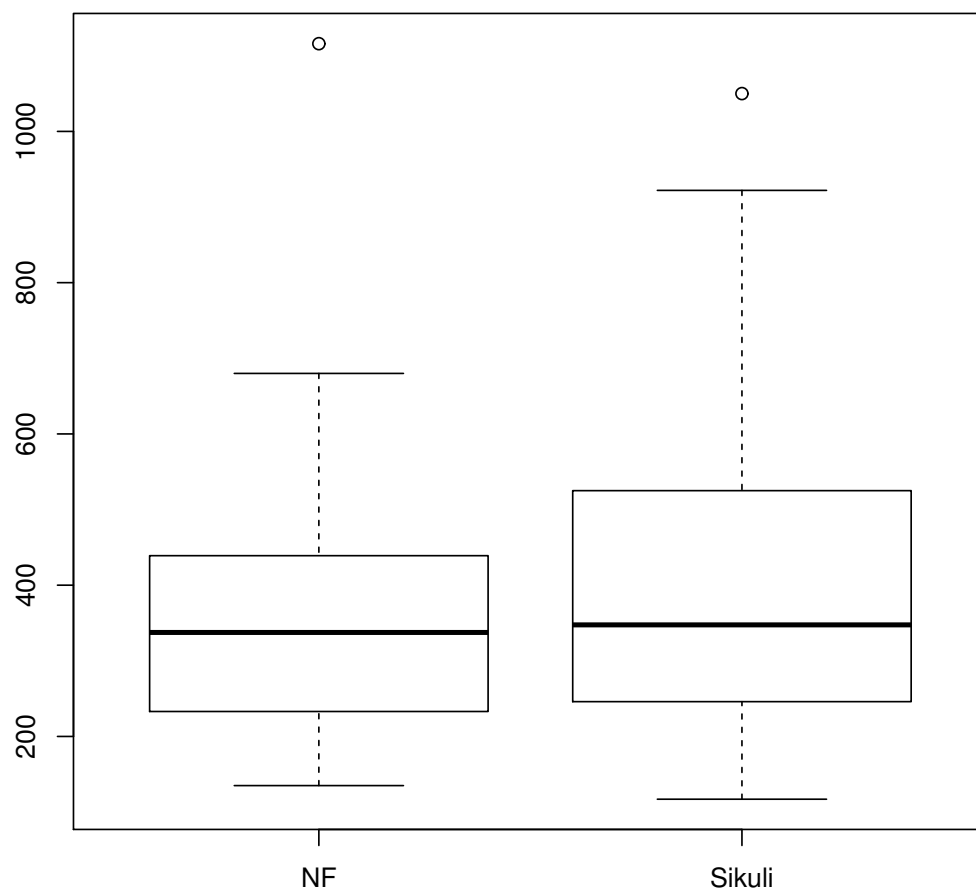


Figure 6.1: Boxplots of the TC-development experiment data samples for NF and Sikuli. The unit is TC development time in seconds.

These outliers and their corresponding paired data points are removed, which leads to the boxplots presented in Figure 6.2. As seen in this figure, the NF data sample now

has one more outlier and the Sikuli data sample now has two outliers:

- TC4 for NF by Subject 2 (679s)
- TC1 for Sikuli by Subject 2 (922s)
- TC28 for Sikuli by Subject 1 (792s).

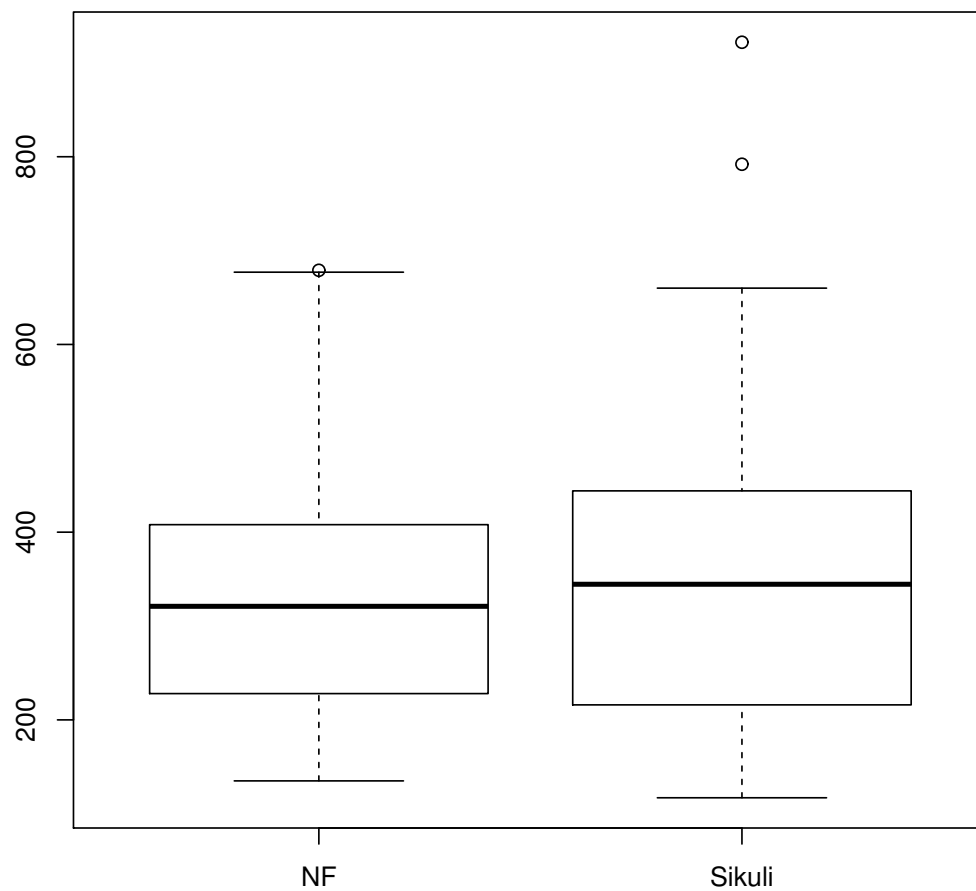


Figure 6.2: Boxplots of the TC-development data samples for NF and Sikuli, after the first paired removal of outliers. The unit is TC development time in seconds.

These outliers and their corresponding paired data points are removed, which leads to the boxplots presented in Figure 6.3. These boxplots don't show any outliers, but

may be considered skewed, positively for the NF data sample and negatively for the Sikuli data sample. The Shapiro-Wilk test applied to these data samples returned the following p-values:

- 0.0001087 for S_{NF} and 0.001775 for S_{Sikuli}
- 0.01525 for S_{NF_p1} and 0.008515 for S_{Sikuli_p1}
- 0.04027 for S_{NF_p2} and 0.07491 for S_{Sikuli_p2} ,

where S_{NF} is the original NF sample, S_{Sikuli} is the original Sikuli sample, S_{NF_p1} is S_{NF} after the first paired removal of outliers, S_{NF_p2} is S_{NF_p1} after the second paired removal of outliers and likewise for S_{Sikuli_p1} and S_{Sikuli_p2} . The values show that, at a 95% confidence interval, the null hypothesis that S_{Sikuli_p2} comes from a normal distribution cannot be rejected, but for S_{NF_p2} such a null hypothesis must be rejected, which means that there is statistical evidence that S_{NF_p2} doesn't come from a normal distribution. Figure 6.4 shows a QQ-plot for S_{NF_p2} in order to try to identify more outliers.

Both QQ plots show some similarity to normal distribution, but especially in the case of S_{NF_p2} many values deviate significantly from a normal distribution. Hence, the analysis continues with the Wilcoxon Matched-pairs Signed-ranks non-parametric test, which gives a p-value of 0.4123. This means that at a 95% confidence interval we fail to reject the null hypothesis that there is no statistically significant difference between NF's and Sikuli's development times of TCs for embedded systems. The conclusion is therefore that there is no statistically significant difference between TC development times using NF and Sikuli. The total development times (after the removal of outliers) are 3:15:02 for NF and 3:35:21 for Sikuli and the mean development times are 05:34 for NF and 06:09 for Sikuli.

6.2.2 TC maintenance

Table 6.2 presents the results of the experiment. Subject 1 registered 20 TC maintenance moments in NF and 20 in Sikuli. Subject 2 registered 21 TC maintenance moments in NF and none in Sikuli. Hence, a total of 41 TC maintenance moments were registered for NF TCs and 20 for Sikuli TCs. Figure 6.5 and Figure 6.6 present histograms for the NF and Sikuli data points, respectively.

MM	Framework	Subject	Maint. Time	TC Name
initial SUT version: version 606				
Test cases created.				
SUT update 1: to version 646				
No maintenance moments carried out.				
SUT update 2: to version 683				
MM ₁	NF	Subject 2	0:56	SD

MM	Framework	Subject	Maint. Time	TC Name
MM ₂	NF	Subject 2	0:57	SE
MM ₃	NF	Subject 2	1:00	SG
MM ₄	NF	Subject 2	1:00	TG
SUT update 3: to version 706				
MM ₅	NF	Subject 1	1:30	SA
MM ₆	NF	Subject 1	1:00	SB
MM ₇	NF	Subject 1	1:00	SI
MM ₈	NF	Subject 1	2:12	TA
MM ₉	NF	Subject 1	3:00	TC
MM ₁₀	NF	Subject 1	2:18	TE
MM ₁₁	NF	Subject 1	2:00	UK
MM ₁₂	NF	Subject 1	1:48	VK
MM ₁₃	Sikuli	Subject 1	2:06	SE
SUT update 4: to version 714				
MM ₁₄	NF	Subject 1	1:00	SK
MM ₁₅	NF	Subject 1	2:12	UB
MM ₁₆	NF	Subject 1	2:18	UH
MM ₁₇	NF	Subject 1	2:30	VD
SUT update 5: to version 717				
No maintenance moments carried out.				
SUT update 6: to version 731				
MM ₁₈	NF	Subject 1	3:00	SF
MM ₁₉	NF	Subject 1	1:00	TD
MM ₂₀	NF	Subject 1	1:24	TF
MM ₂₁	NF	Subject 1	1:48	UF
MM ₂₂	NF	Subject 1	1:30	UG
MM ₂₃	Sikuli	Subject 1	1:48	SJ
MM ₂₄	Sikuli	Subject 1	3:00	VJ
SUT update 7: to version 746				
MM ₂₅	NF	Subject 2	0:21	SD
MM ₂₆	NF	Subject 2	0:25	TH
MM ₂₇	NF	Subject 2	0:22	TK

MM	Framework	Subject	Maint. Time	TC Name
MM ₂₈	NF	Subject 2	0:32	UI
MM ₂₉	NF	Subject 2	0:34	UJ
MM ₃₀	NF	Subject 2	0:34	VJ
MM ₃₁	Sikuli	Subject 1	2:30	TK
SUT update 8: to version 754				
MM ₃₂	NF	Subject 1	1:48	VG
MM ₃₃	Sikuli	Subject 1	3:36	SG
SUT update 9: to version 761				
MM ₃₄	NF	Subject 1	2:00	SC
MM ₃₅	Sikuli	Subject 1	2:12	SD
MM ₃₆	Sikuli	Subject 1	2:48	TH
MM ₃₇	Sikuli	Subject 1	2:06	TI
MM ₃₈	Sikuli	Subject 1	3:48	UC
MM ₃₉	Sikuli	Subject 1	3:12	UJ
MM ₄₀	Sikuli	Subject 1	2:06	VF
MM ₄₁	Sikuli	Subject 1	2:18	VH
SUT update 10: to version 780				
MM ₄₂	Sikuli	Subject 1	2:24	TG
MM ₄₃	Sikuli	Subject 1	2:06	UE
MM ₄₄	Sikuli	Subject 1	2:00	VA
MM ₄₅	Sikuli	Subject 1	2:00	VB
MM ₄₆	Sikuli	Subject 1	1:48	VE
SUT update 11: to version 792				
MM ₄₇	NF	Subject 1	1:06	TJ
MM ₄₈	NF	Subject 2	0:15	SD
MM ₄₉	NF	Subject 2	0:16	SE
MM ₅₀	NF	Subject 2	0:19	SG
MM ₅₁	NF	Subject 2	0:27	UC
MM ₅₂	NF	Subject 2	0:27	UD
MM ₅₃	NF	Subject 2	0:41	VA
MM ₅₄	NF	Subject 2	0:22	VB
MM ₅₅	NF	Subject 2	0:26	VE

MM	Framework	Subject	Maint. Time	TC Name
MM ₅₆	NF	Subject 2	0:27	VF
MM ₅₇	NF	Subject 2	0:26	VH
MM ₅₈	NF	Subject 2	0:26	VI
MM ₅₉	Sikuli	Subject 1	4:18	UI
MM ₆₀	Sikuli	Subject 1	2:06	VI
SUT update 12: to version 815				
No maintenance moments carried out.				
unknown SUT version				
MM ₆₁	Sikuli	Subject 1	3:00	UD

Table 6.2: The results of the TC-maintenance experiment. MM is Maintenance Moment. Framework is the framework used. Subject is the subject that carried out the maintenance and Maint. Time is the time the maintenance took. TC Name is the name of the TC that maintenance was carried out on. TC names marked in yellow, orange and red required two, three and four maintenance moments, respectively.

Analysis

The data point MM₆₁, as presented in Table 6.2, was obtained in connection to an unknown SUT-version update. It will therefore be regarded as an erroneous data point and disregarded from further analysis.

Figure 6.7 shows boxplots of both samples obtained in the TC-maintenance experiments. As seen in the boxplots, nearly all the Sikuli data points lie above the 3rd quartile of the NF data sample, which means that the Sikuli values are generally higher than the NF values. It also looks like the NF sample may come from a normally distributed population, while the Sikuli data sample seems to be strongly skewed to the right (positive skewness). The Shapiro-Wilk test gives the p-values of 0,0006725 for NF and 0,004513 for Sikuli, which means that the null hypotheses that these samples come from normally distributed populations both have to be rejected at a 95% confidence interval, so there is enough statistically significant evidence to say that the samples do not come from normally distributed populations, so a non-parametric test will be used.

Since the samples are not paired in this experiment, the Mann-Whitney U-test is applied. The test gives a p-value of 0,000002027, so the null hypothesis has to be rejected at a 95% confidence interval. The alternative hypothesis has to be accepted, that the location shift between the distributions of the samples are different from 0. Hence, there is statistically significant evidence that there is a difference in TC-maintenance times between NF and Sikuli. The boxplots in Figure 6.7 lead to the conclusion that TC maintenance is faster in NF than in Sikuli.

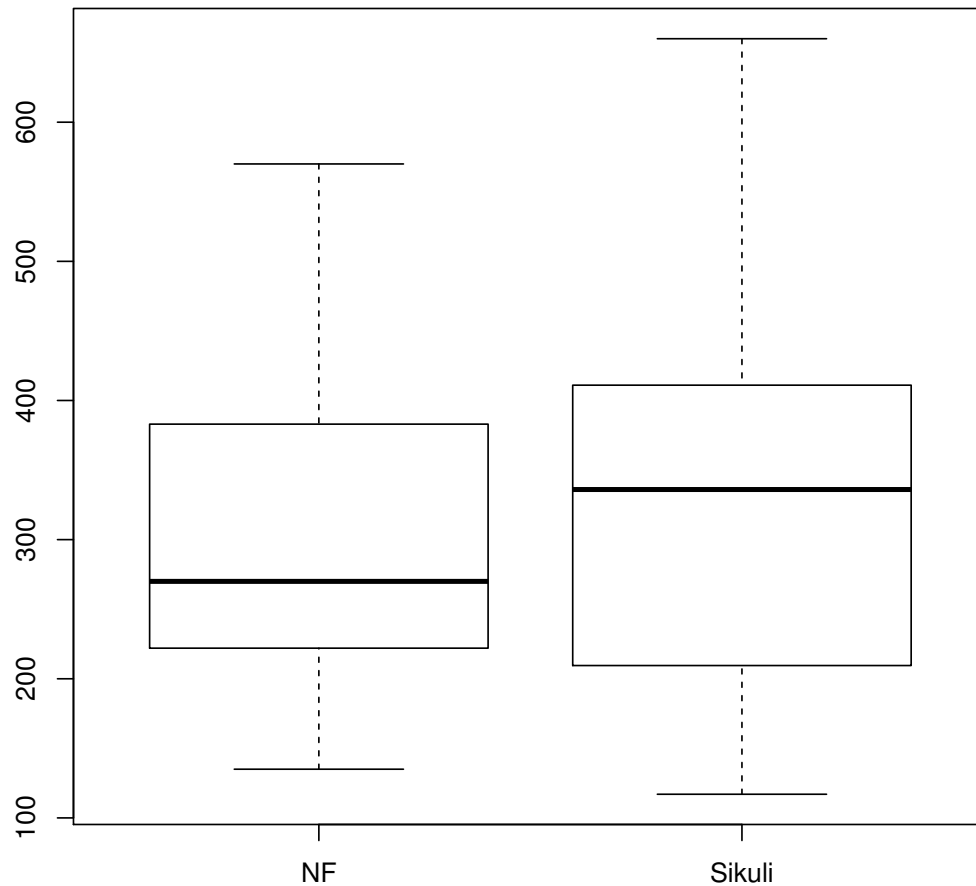


Figure 6.3: Boxplots of the TC-development data samples for NF and Sikuli, after the second paired removal of outliers. The unit is TC development time in seconds.

6.2.3 Qualitative data

This section presents the qualitative data collected in the experiments.

TC development

NF's automated test-bench generation shortened the time required to write the skeleton of the test suite and the code that initialises the SUT. For the SUT used in the experiments it took 26:52min to write this code in Sikuli and 7:27min in NF. For a SUT with

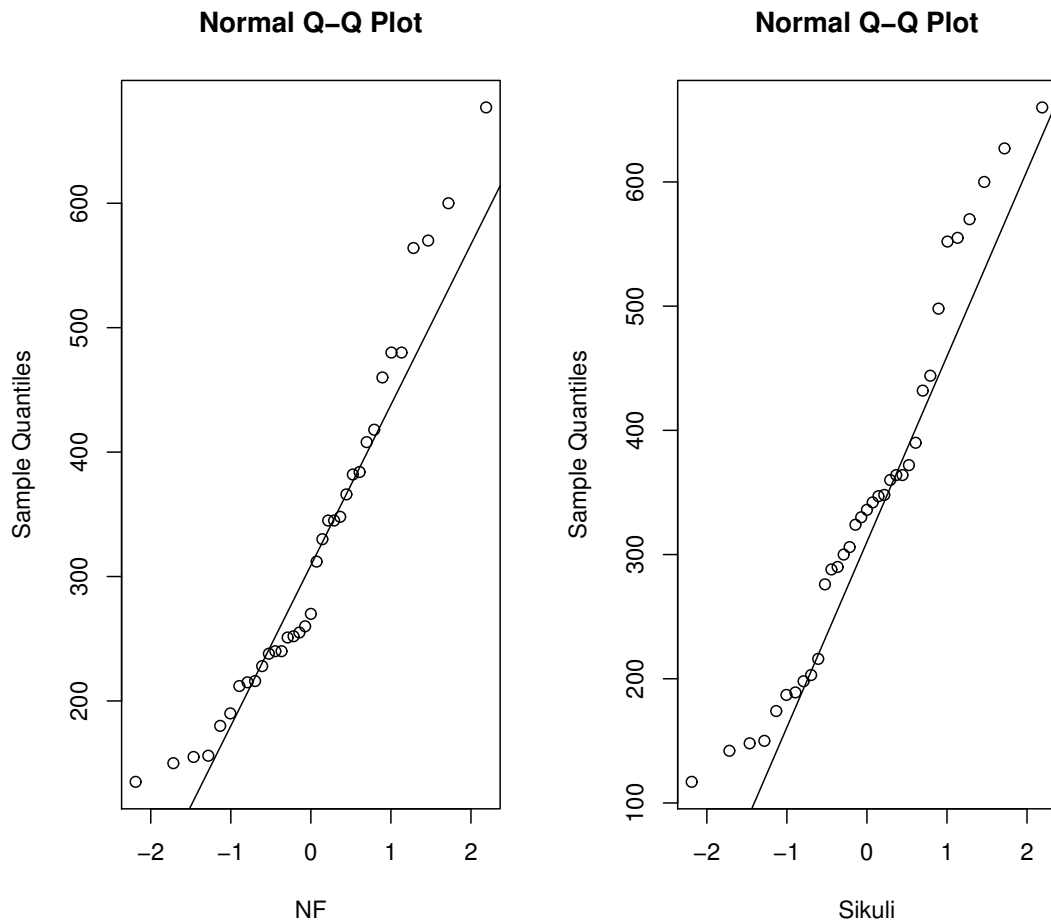


Figure 6.4: QQ-plot diagrams of the TC-development data samples for NF and Sikuli.

many TCs, however, this time difference would be amortised throughout the total time needed to develop many TCs and would be less and less significant per developed TC.

Both frameworks were perceived to perform equivalently well in TC development. Notably, NF's way to obtain reference images provided the possibility to obtain them for many TCs in parallel. An advantage of Sikuli's way was the possibility to select particular regions of the SUT's GUI. For NF, if an old reference image existed, after each run the framework saved an already cropped most similar image in the last_run directory. If a reference image did not exist, however, NF always saved a screenshot of the SUT's entire GUI and an external program had to be used to crop the acquired image to the region of interest.

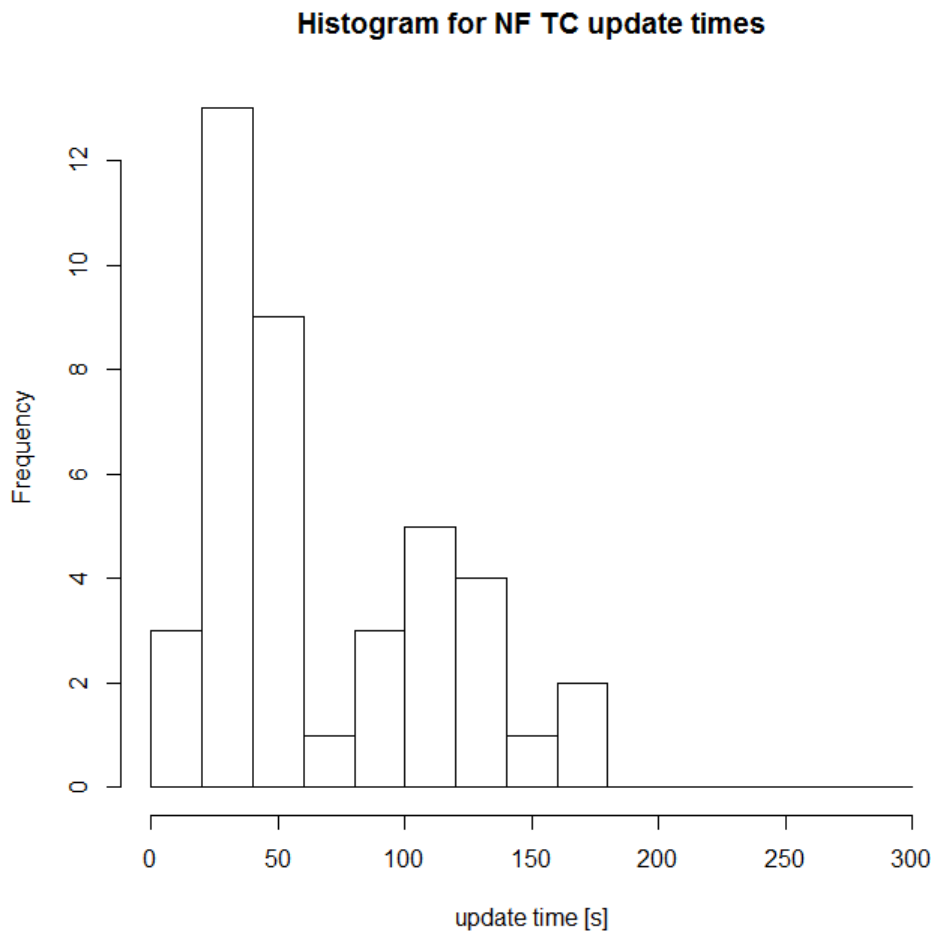


Figure 6.5: A histogram of the data points collected for TC maintenance in NF. It shows the number of maintenance moments that took a certain amount of time to carry out.

TC maintenance

It was observed that TCs failed either because of small rendering differences from run to run, or because the SUT's behaviour or GUI changed in an update. The later case rendered a total of 61 occurrences of failing TCs, which became subject to analysis in this experiment. The type of changes discovered by the TCs were mostly slight changes to the GUI (e.g. lighter text), but in some cases whole elements disappeared from the GUI. In one case the SUT no longer reacted to button being long-pressed. The changes discovered by the TCs were similar to the ones discovered by the Company practitioners with manual tests.

The TC maintenance processes differed between NF and Sikuli. Since NF included the expected, actual and difference images in its reports, it was easy to see why a

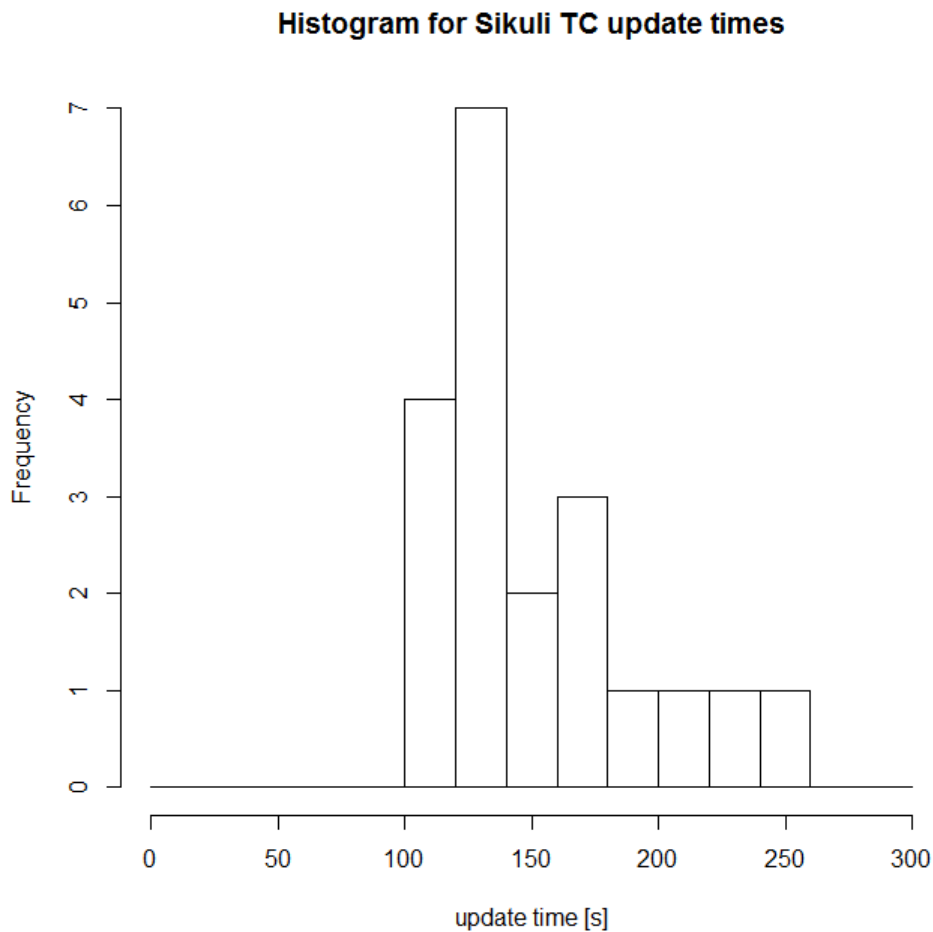


Figure 6.6: A histogram of the data points collected for TC maintenance in Sikuli. It shows the number of maintenance moments that took a certain amount of time to carry out.

certain TC failed in NF. Also since a scalar value of similarity between the images was presented, it was easy to adjust the sensitivity of the image comparison algorithm accordingly. NF always presented a similarity value, even if the images were totally different. Image comparisons and similarity calculations did not take long as perceived by the experimental subjects. It was easy to update reference images, since new "actual" images were available after each test run. In Sikuli it was difficult to find the reason for failed image comparisons because, as shown in Figure 6.8, the only information produced by Sikuli for failing image comparisons was that the image could not be found. In such cases the TC was rerun and carefully watched for anomalies. In some cases the TC needed to be rerun several times or run manually in order to give the tester the time to investigate why a certain image comparison failed. It was difficult to adjust the sensitivity of Sikuli's image comparison algorithm, because no reference was provided

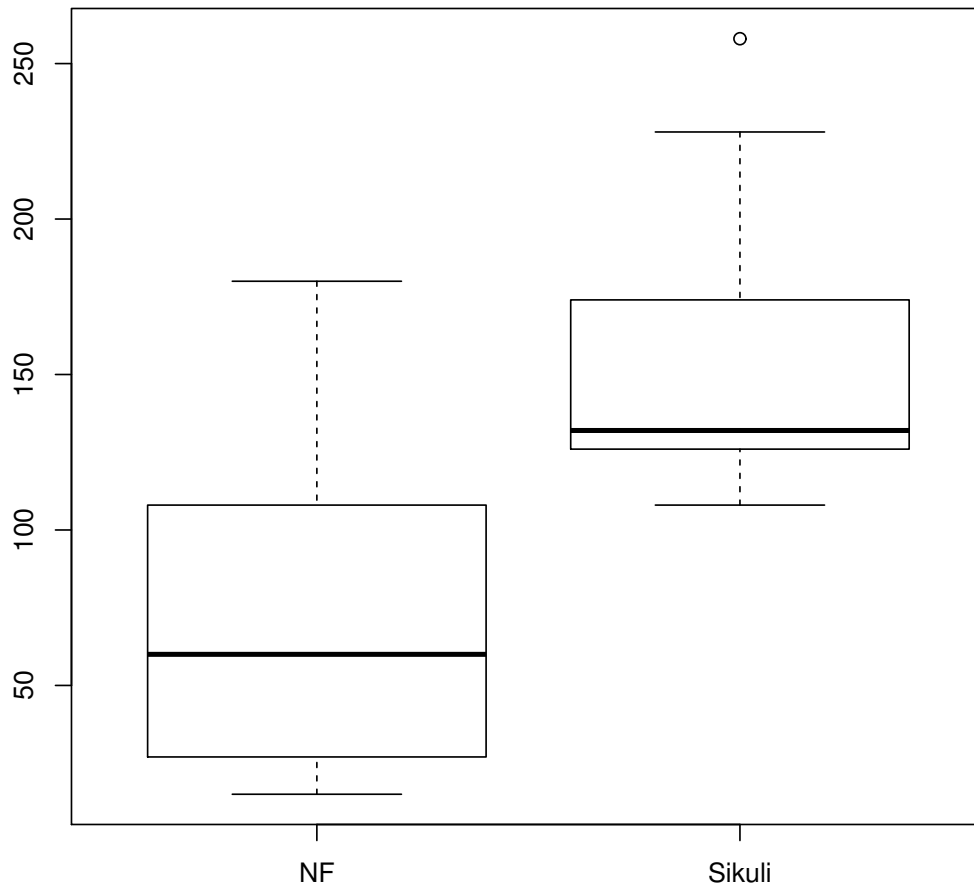


Figure 6.7: Boxplots of the data samples obtained in the TC-maintenance experiments for NF and Sikuli. The unit of the data points is the time (measured in seconds) it took to carry out an update of a failing TC in the respective framework.

in form of how close the framework was to finding a match. This also poses the risk that the sensitivity be set too low, which can introduce false positive test results. It was more difficult to update reference images in Sikuli than in NF, because in Sikuli the SUT needed to be rerun and the GUI traversed manually until a new screenshot could be taken.

TCs failed more often in NF (41 failures) than they did in Sikuli (19 failures that were connected to a particular SUT update). In two cases the same test failed in both frameworks after the same SUT update; the rest of failures were disjoint. The reason

for this could be the robustness of the frameworks' image comparison algorithms in the context of changes that occurred in the SUT's GUI, where both frameworks missed the SUT change that the other framework's image comparison algorithm reported as different. For NF it was noted that there was no big difference between the similarity values of failed and passed TCs, which indicates that NF's image comparison algorithm may have been a bad match for the SUT's context. As for Sikuli, since it did not report any value for the similarity between the compared images, it could be that the tolerance used in the TCs was too high, which made Sikuli miss certain changes. Finally, the fact that the SUT often rendered the GUI differently could have made it more difficult for the image comparison algorithms to discover relevant changes in the SUT's GUI.

MyTitle

Start Time: 2015-10-06 19:41:32

Duration: 0:00:09.953000

Status: Error 1

This is a description.

Show [Summary](#) [Failed](#) [All](#)

Test Group/Test case	Count	Pass	Fail	Error	View
UnitTestX	1	0	0	1	Detail
test_experiment			error		<div style="border: 1px solid gray; padding: 5px;"> <pre> ftl.1: Traceback (most recent call last): File "C:\Users\MACIEJ-1.MAK\AppData\Local\Temp\sikuli-930229594167568213.py", line 23, in test_experiment click("1397740462222.png") FindFailed: FindFailed: can not find 1397740462222.png on the screen. Line 1574, in file Region.java </pre> </div>
Total	1	0	0	1	

Figure 6.8: A failed image comparison as reported by Sikuli.

Conclusion	Description
EXP₁	There is no statistically significant difference in TC development times between NF and Sikuli.
EXP₂	The total TC development times in the experiment were 3:15:02 for NF and 3:35:21 for Sikuli and the mean TC development times were 05:34 for NF and 06:09 for Sikuli.

EXP₃	<p>Out of the total of 60 TC failures that were connected to a particular SUT update, two TCs failed in both frameworks for the same SUT version, while the rest of failures were disjoint. The reasons for this could be:</p> <ul style="list-style-type: none"> • Low robustness of the frameworks' image comparison algorithms in the context of the SUT • Inaccurately set tolerance for Sikuli's image comparison algorithm • The fact that the SUT's GUI was often rendered differently from run to run.
EXP₄	TC maintenance is statistically significantly faster in NF than Sikuli.
EXP₅	NF's automated test-bench generation shortened the time required to write the SUT initialisation and test suite skeleton code, but this time is amortised by the time needed to create many TCs and is less significant per TC in the long run.
EXP₆	Sikuli's and NF's processes to obtain reference images performed equivalently well, but NF's process made it possible to obtain reference images for multiple TCs in parallel.
EXP₇	It was easier to obtain reference images for regions of the GUI in Sikuli, as it let the tester select the region of interest for each screen capture. In order to achieve a similar result in NF, an external image editing program had to be used.
EXP₈	TCs failed either due to irrelevant differences in how the Engine rendered images, or due to updates to the SUT that changed its behaviour or GUI.
EXP₉	The TCs discovered a total of 61 changes to the SUT. The changes discovered were: missing, added or changed parts of the GUI, as well as the fact that the SUT no more reacted to a long-pressed button. The discovered changes were similar to the ones discovered with manual tests.
EXP₁₀	NF's test run reports with the expected, actual and difference images made it easier to find the reason for failing image comparisons.
EXP₁₁	NF's scalar difference value presented for each image comparison made it easier to adjust the image comparison algorithm's sensitivity in NF.
EXP₁₂	The way reference images were updated was less time-consuming in NF than it was in Sikuli.

Table 6.3: A summary of the results of the conducted experiments.

7

Synthesis

This section synthesises and discusses the results of the particular phases of this study and is structured after the study's research questions.

7.1 VGT of embedded systems

The Company's typical embedded systems provide a visual interface, which opens up for VGT. The prestudy showed that there is need for a better approach than manual for testing of the visual interface and VGT is a possible candidate (CS₁). The experiments confirmed that VGT was meaningful for the Company's embedded SUTs (CS₁₁, EXP₉). *This shows that VGT is beneficial for embedded systems.*

During the TC maintenance experiment 61 failing TCs showed changed SUT behaviour after an update. The reported changes were: missing, added and changed elements of the SUT's GUI, as well as the fact that the SUT no more reacted on a long-pressed button. These changes are similar to the ones discovered with manual tests (EXP₉). *This indicates that VGT could be a possible complement for manual testing of embedded systems.*

The study found that non-standard interfaces such as the FU-HMI interface should be tested for achieving a satisfactory level of test coverage (CS₈, CS₁₁). *For the case of embedded systems in general these observations indicate that VGT of such systems should take into account their specific I/O interfaces.*

Even though Sikuli could not send hardware control events to the SUT (CS₁₄), this behaviour could be simulated by visual components added to the SUT's GUI (CS₁₂). NF however supported the hardware controls interface natively (NF₂). *These observations show two ways of VGT of embedded systems with non-desktop I/O interfaces. One way is to make desktop-compatible parts of the SUT's interface simulate the non-compatible parts and then use a desktop VGT framework to test the system. Another way is to give the testing framework native support for such non-desktop interfaces.*

Based on the Company's requirements (CS₆, CS₇, CS₈, CS₉, CS₁₀) and the functionality that Sikuli provides (CS₁₂, CS₁₃, CS₁₄), Sikuli is missing native communication through the hardware control interface and through the FU interface in order to fulfill the Company's requirements fully. Sikuli's reports would also need to provide more information, especially present the expected, actual and difference images and a scalar value of similarity in order to fulfill the Company's requirements. It is left as a future research suggestion to investigate how much work it would require to adapt Sikuli for VGT of embedded systems. In this study's context one difficulty has been identified with that in form of Sikuli's lack of support for the ctypes python library, which is required in order to reach the native FU and hardware control interfaces of the SUT. *These results indicate that in order for a framework to be suitable for VGT of an embedded system, it is important that this framework be able to reach this embedded system's specific I/O interfaces. Also VGT test run reports should present the expected, actual and difference images, as well as report a similarity value for the images. More research would be needed to see how much effort is required to adapt a desktop VGT tool for testing of embedded systems in general.*

7.2 Development cost of VGT TCs for embedded systems

Sikuli's and NF's processes for TC development performed equivalently well (EXP₁). NF however makes it possible to capture multiple reference images in parallel, which may make its process more efficient (EXP₆). On the other hand, with Sikuli, regions of the GUI can be selected easily, while in NF external image editing programs need to be used to achieve this (EXP₇). The total TC development times in the experiment were 3:15:02 for NF and 3:35:21 for Sikuli and the mean TC development times were 05:34 for NF and 06:09 for Sikuli (EXP₂). *More experiments would be needed to confirm this, but the conclusion is that the most efficient process for a framework to obtain reference images may be a combination of NF's and Sikuli's processes, where the framework always stores screenshots of the GUI from the most recent test run and if desired by the tester, such screenshots can be used as future reference images with the possibility to crop them beforehand.*

The test-bench generation functionality helped shorten the time required to write SUT initialisation and test suite skeleton code. This gain is, however, amortised throughout the whole test suite for a given SUT, so its significance decreases in tact with the amount of tests being created (EXP₅). *The conclusion is that automatic test-bench generation is beneficial for TC development costs, but that such benefit would be the less significant the more TCs would be written for a particular SUT.*

7.3 Maintenance cost of VGT TCs for embedded systems

TC maintenance was significantly faster in NF than Sikuli (EXP₄). *This indicates that to lower TC maintenance costs a process should be preferred where the framework stores screenshots of the SUT from the most recent test run that the tester can choose to use as*

future reference images rather than a process where the SUT has to be re-run and new images captured manually.

The information in NF's test run reports made it easier to see why a certain TC failed (EXP₁₀), which sped up TC maintenance. *The conclusion is that for faster TC maintenance the test run reports should contain as much information as possible about the SUT's state upon the failure. Especially for failed image comparisons the report should present the expected, actual and difference images of the GUI.*

TCs often failed due to irrelevant differences in how the Engine rendered the GUIs from test run to test run (EXP₈). During the experimenters the sensitivity of the image comparison algorithms was decreased to deal with such cases. It was easier to find the desired sensitivity in NF, since it reported a scalar similarity value for each image comparison, which served as a reference point to set the sensitivity for future test runs. (EXP₁₁). *The conclusion is that in order to make it easier to tune the sensitivity of the framework's image comparison algorithm, the testing framework should report a value on how close it was to a match for image comparisons.*

Irrelevant differences in the GUI can also be disregarded by using a custom image comparator that is only sensitive to specific types of differences, provided that the framework supports usage of external image comparators. *The conclusion is that the testing framework should allow usage of custom image comparators to make it sensitive only to the types of differences that it should capture.*

Out of the total of 60 TC failures registered during the TC maintenance experiment two TCs failed in pairs among the two frameworks for the same SUT update, while the remaining 56 TC failures were disjoint among the SUT update and TC. The possible reasons that TC failures did not always come in pairs among the two frameworks were: low robustness of the frameworks' image comparison algorithms in the context of the SUT, inaccurately set tolerance for Sikuli's image comparison algorithm and the fact that the SUT's GUI was often rendered differently from run to run (EXP₃). *The conclusion is that it is important for the image comparison algorithm used by the framework to be robust in the context of the SUT.*

8

Validity Threats

A validity threats analysis was done based on the check-list provided by Wohlin et al. [43, pp. 104-110]. Wohlin et al. explain that designing a study in a certain way may increase one type of validity, but at the same time decrease another [43, pp. 111]. Following Wohlin et al.'s classification, this study was applied research. According to the authors' recommendations for applied research studies, validity threats were prioritised in the following order:

- Internal validity
- External validity
- Construct validity
- Conclusion validity

8.1 Internal validity

A possible threat to the experiment results' internal validity is that the results were caused by other factors than the choice of the testing framework. In order to mitigate that risk, a list of potential disturbing factors was established and an experiment design was chosen that would minimise their effect on the results. For instance, the difference in the experimental subjects' work experience could affect the results. If some TCs were simpler to implement or maintain, this could also affect the results. A randomised paired comparison design [28, section 5.3.2] was used for the TC development experiment, where treatments, subjects and objects were assigned to each other at random, which should minimise the effect of the above disturbing factors on the results.

The default random-number naming scheme was used for new reference images in Sikuli. This means that, even though the names of missing images were reported by Sikuli, such names did not provide any direct association for the tester to an actual

image. It is possible that if meaningful names were given to reference images, it would have been easier to identify failing image comparisons in Sikuli, which could lead to shorter maintenance times.

In the TC maintenance experiment there were 56 TC failures that did not occur in pairs, i.e. a TC failed in one of the frameworks, but not the other. This means that the TC did not fail in one of the frameworks even though the SUT had changed, which is a false negative result reported by that framework. Since no failure was reported by such a framework, no maintenance was done to the TC, but it is possible that if the image comparison algorithms had better sensitivity to the type of changes that occurred in the SUT, and the TC failed in both frameworks, this would have generated more paired data points and different experiment results.

8.2 External validity

Since the study was conducted at only one company that operates within one domain, there is the risk that its findings are not applicable to embedded systems in general. Based on the results of the particular phases of this study and the knowledge of the author of this report about embedded systems in general, deductive logic was used to reach conclusions about embedded systems in general whenever such generalisation could be done. This means that even though not all results of this study may have been possible to generalise, whenever such generalisation was done, the external validity of the resulting findings should be high.

Similarly this study was only conducted using NF and Sikuli and it is possible that other tools would yield different results. From the point of view of VGT TC development and maintenance costs, this study identified two differentiators between these two frameworks. While their influence on VGT TC development and maintenance costs should be the same in other VGT frameworks, it is possible that this study would reveal other factors that make a difference in VGT TC development or maintenance costs if repeated with other VGT frameworks. For this reason, but also in order to increase the conclusion validity of this study's findings, this study should be repeated with other VGT frameworks.

There is a risk that the profile of the experimental subjects differs significantly from the one of industrial practitioners, which would be an example of interaction of selection and treatment and a risk for the generalisability of the results. Since both subjects were qualified for working with the Company's SUTs, this risk is judged to be low. This judgement is further justified by Höst et al., who conclude that, under certain conditions, software engineering students may be used in empirical studies instead of professional software developers [26].

The experimental objects used could be non-representative for typical TCs and SUTs used for the Company's embedded systems or for embedded systems in general (interaction of setting and treatment). To mitigate this risk, the experimental objects were chosen based on knowledge from the prestudy about typical systems and TCs at the Company. The Company's employees have also verified the representativeness of the ex-

perimental objects. Deductive logic was used for evaluating the representativeness of the experimental objects for arbitrary embedded systems and TCs. The conclusion is that as long as the Company's SUTs' I/O interfaces are representative for other embedded systems, this study's results should still be valid for such systems.

8.3 Construct validity

There is a risk that the interviewers and the interviewees interpreted the interview questions differently. Two interviewers and two interviewees participated in the interview and discussions were encouraged, so inclarities were likely to be captured and resolved. On the other hand, the fact that as many as four people participated in the interview could induce evaluation apprehension and lead to misleading answers from the interviewees.

There is a risk that the interviewees had certain expectations or hopes connected to any of the testing frameworks, which biased their answers.

The variables measured in the experiments could have been a bad measure for what was intended to be studied. The hypothesis was that certain characteristics of a testing framework influence the cost of TC development and maintenance. Two different frameworks were alternatives in the experiment and the measured outcome was TC development and maintenance times. This setup is judged as representative for the tested theory.

There is a risk that the TCs defined for the experiments were not representative for typical TCs at the Company. They were, however, defined for a commercial SUT and based on:

- a qualitative analysis of the SUT's requirement specification
- a qualitative analysis of a test suite written for the SUT for the old framework
- observations of the process of manual testing at the Company.

This risk is therefore judged to be low.

There is also a risk that the process for TC maintenance data points generation was not representative for TC maintenance as seen in industry. While traditionally TC maintenance originates from a change in the requirements for a SUT, TC maintenance moments in the experiments were based on changed SUT behaviour rather than changed requirements. The construct validity of these results is therefore dependent on the accuracy of the root-cause analysis done by the experimental subjects for failing TCs and on their knowledge of the requirements for the SUT.

If the experimenters had any expectations or hopes on whichever framework would perform better in the experiments, they could have unknowingly influenced the treatment and hence the results. Since the experimenters were the creators of NF, such bias would expectedly render shorter TC development and maintenance times for NF.

Furthermore, since the experimenters were the creators of NF, there is the risk that they were more familiar with NF, which could lead to shorter TC development and maintenance times in NF. This risk is judged low however, as the experimenters ran the

experiments after the case study where they got familiar with how to write and maintain TCs in Sikuli.

8.4 Conclusion validity

Deductive logic was used to draw conclusions about RQ1 from the prestudy and the experiments. The multiple sources of information provided triangulation in order to achieve high conclusion validity. For RQ2 and RQ3 also quantitative data was used. There is a risk that the significance levels used in the experiments were too low. They were however chosen according to wide-spread standards used in academia. Also the tests used for statistical analysis were chosen based on the type of data being studied.

There is a risk that each of the experimental subjects applied treatment differently (reliability of treatment implementation), which could distort the results. To minimise this risk, a protocol was established in the experiment design phase for how to apply treatment. The experiments were also designed to minimise the impact of potential differing treatment on the results.

9

Conclusion

This thesis evaluated VGT of embedded systems. The goal was to provide guidelines for how to implement VGT of embedded systems with focus on TC development and maintenance costs. This chapter presents the study's findings grouped by the research questions. The chapter ends with future research suggestions.

9.1 RQ1: VGT of embedded systems

RQ1: To what degree is VGT applicable for embedded systems in industrial practice?

The study found VGT to be fully applicable and beneficial for testing of embedded systems. As shown in the experiments, the technology finds similar defects as manual testing, but it also has the potential to make visual testing more effective and efficient compared to manual testing, which leads to reduced cost or improved quality. This is a contribution to industrial practitioners in form of a use case for VGT of embedded systems. Since there is not much literature about script-driven automated GUI testing[33], this thesis is a contribution to academia. Also since no studies have been found about applicability of VGT to embedded systems, this is a contribution to academia and industrial practitioners per se.

The thesis found that VGT for embedded systems needs to take into consideration such systems' specific I/O interfaces in order to be successful. This study lists two ways of VGT of embedded systems whose I/O interfaces differ from those of desktop systems; One way is to make desktop-compatible parts of the SUT's interface simulate the non-compatible parts and then use a desktop VGT framework for testing. Another way is to make the testing framework support such non-desktop interfaces natively.

The study concluded that Sikuli did not fulfill the Company's requirements for testing of their embedded systems fully because of the lack of support for the FU interface. More

research would be required to find the effort needed to make a desktop VGT tool suitable for testing of embedded systems.

Further guidelines found by this thesis are connected to TC development and maintenance and can be found in the following two sections.

9.2 RQ2: TC development costs

RQ2: What are the development costs associated with VGT TCs for embedded system environments in industrial practice?

The total TC development times in the experiment were 3:15:02 for NF and 3:35:21 for Sikuli and the mean TC development times were 05:34 for NF and 06:09 for Sikuli. This difference was not found to be statistically significant. Still, the thesis proposes an improved TC development process based on the experiments' qualitative results. The suggestion is to combine NF's and Sikuli's processes for how to obtain reference images, where the framework stores screenshots of the SUT's GUI from the most recent test run and when desired by the tester, offers the possibility to crop them and use as future reference images. More experiments would be needed to test this process, but it should be seen as a contribution to current body of knowledge as it can potentially reduce VGT TC development costs. Furthermore, no factors have been found that would suggest that this result be inapplicable to other systems than embedded systems, so this process could potentially be implemented for VGT of other systems than embedded.

The TC development times obtained in the experiments are a contribution per se as a metric for TC development cost. Since the experiment started from TC descriptions in natural language, the results are also input to Alégroth et al.'s discussion about the effort needed for transision of manual TCs to VGT TCs [3].

Another conclusion is that automatic test-fixture and example-TC generation is beneficial for TC development costs, but that such a benefit tends to be the less significant the more TCs are created for a particular SUT. No factors have been found that would suggest that this result be inapplicable to other systems than embedded.

9.3 RQ3: TC maintenance costs

RQ3: What are the maintenance costs associated with VGT TCs for embedded system environments in industrial practice?

The study has found TC maintenance to be significantly less costly in NF than in Sikuli. The main differentiators between the two frameworks were:

- the amount and type of information in the test run reports
- the TC maintenance process.

Therefore these are the supposed main contributing factors to the obtained difference in TC maintenance costs. The recommendation is therefore to use a TC maintenance process, where the framework stores screenshots of the SUT from the most recent test run that the tester can use as future reference images rather than a process where the SUT has to be re-run and new images re-captured manually. This conclusion is a valuable guideline for anyone who wants to develop a framework for testing of embedded systems. No factors have been found that would suggest that this result be inapplicable to other systems than embedded.

Another conclusion is that test run reports should contain as much information as possible about the SUT upon test failures to help the tester evaluate the correctness of test failures. Especially failed image comparisons should be reported with the expected, actual and difference images. Also images' similarity should be reported as a reference point to make it easier to tune the sensitivity of the image comparison algorithms. Furthermore, a VGT framework should allow usage of external image comparison algorithms to make it sensitive only to the type of GUI differences of interest. These guidelines are a contribution to the need expressed by Alégroth et al. for a better technique for verifying the correctness of VGT TCs [3]. The guidelines also confirm what was stated by Meyer et al., that the framework should keep record of failed test scenarios [33]. No factors have been found that would suggest that these guidelines be inapplicable to other systems than embedded.

The TC maintenance times obtained in the experiments contribute to Alégroth et al. and Börjesson and Feldt, where the authors report the need of further research within the subject of maintenance costs of automated test suites [3, 8].

Finally it was observed that in order for the testing framework to detect as many changes as possible in the SUT, it is desirable for the framework's image comparison algorithms to be as robust as possible in the SUT's context.

9.4 Future research suggestions

Neither Sikuli, nor NF supported verification of animations. Especially states where the SUT presents an endless animation could not be verified by the frameworks, since each time a screenshot of the SUT was captured, the GUI looked different, which led to a TC failure. This subject, even though out of scope for this thesis, was recognised as a need at the Company and is worth further research, including its impact on current TC development and maintenance processes and costs. The same suggestion applies to the need for comparisons of non-rectangular areas of GUIs.

TCs used in this study sometimes failed due to GUI differences irrelevant from the testing perspective. In order to handle this, the image comparison algorithms' sensitivity was decreased, but this suggests an interesting research topic on guidelines on how to implement a robust image comparison algorithm for a given VGT context.

Finally, this study concluded that Sikuli was missing certain features to able to offer a satisfactory level of test coverage. It would be interesting to see how much effort would be required to adapt an open-source VGT tool for testing of embedded systems in a given

context. It would also be profitable to replicate this study with other VGT tools than Sikuli such as eggPlant Functional or JAutomate in order to give more support for the findings of this study.

A

Interview questions

The following questions were asked during the interview described in section 4.1. The questions were designed to be open-ended and allow for follow up questions:

- How do you do HMI testing today?
- What does the testing workflow look like?
 - Who creates the FU mock-ups?
 - How often do FU interfaces change throughout a project?
- How would you like to use a new testing framework? Regression testing? Test-driven development?
- Have you used the existing testing framework?
- How would you like to create and organise your test suites and test cases?
- How would you like to be able to interact with the HMI?

B

TC template

Table B.1 represents the template used for defining test cases in natural language. Every TC defined consisted in one to several stimuli sent to the SUT and then exactly one verification performed. Apart from specifying the stimuli sent to the SUT and the verification performed, the template also specifies an ID (TC id), a short description (TC goal) and a longer description (TC description) of a TC. Moreover, the covered types of input and output interfaces to/from SUT are listed.

TC id	<TC identifier>
TC goal	<short description of this TC's goal>
TC description	<longer description of this TC>
stimulus 1	<stimulus 1 to SUT>
stimulus 2	<stimulus 2 to SUT>
...	...
stimulus n	<stimulus n to SUT>
verification	<verification of SUT's behaviour>
touch screen	<is touch screen involved?>
hardware controls	<are hardware controls involved?>
sending indications, data and events	<are indications, data and events involved?>
verifying screen output	<is verifying screen output involved?>
receiving actions	<is receiving actions involved?>

Table B.1: The template for defining test cases.

Bibliography

- [1] E. Alégroth. On the industrial applicability of visual gui testing. *Department of Computer Science and Engineering, Software Engineering (Chalmers), Chalmers University of Technology, Goteborg, Tech. Rep*, 2013.
- [2] E. Alégroth. *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. PhD thesis, Chalmers University of Technology, 2015.
- [3] E. Alégroth, R. Feldt, and H. Olsson. Transitioning manual system test suites to automated testing: An industrial case study. In *The 6th International Conference on Software Testing, Verification and Validation*, 2013.
- [4] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. *Computer Aided Verification: 10th International Conference, CAV'98 Vancouver, BC, Canada, June 28 – July 2, 1998 Proceedings*, chapter MOCHA: Modularity in model checking, pages 521–525. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-69339-0. URL <http://dx.doi.org/10.1007/BFb0028774>.
- [5] E. Alégroth, M. Nass, and H. Olsson Holmström. Jautomate: A tool for system- and acceptance-test automation. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, pages 439–446, 2013. ISBN 978-0-7695-4968-2.
- [6] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, volume 404. Springer US, Boston, MA, 1997. ISBN 9781461561279;1461561272;1461378087;9781461378082.
- [7] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003. ISSN 0018-9162.

- [8] E. Börjesson and R. Feldt. Automated system testing using visual gui testing tools: A comparative study in industry. In *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation*, 2012.
- [9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ACM SIGSOFT international symposium on Software testing and analysis*, volume 27, pages 123–133, July 2002.
- [10] C. Brooks, E. Lee, and S. Tripakis. Exploring models of computation with ptolemy ii. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 331–332, Oct 2010.
- [11] I. Burnstein. *Practical software testing: a process oriented approach*. Springer, New York, 2003. ISBN 0387951318;9780387951317;.
- [12] K. Chakrabarty. Modular testing and built-in self-test of embedded cores in system-on-chip integrated circuits. In R. Zurawski, editor, *Embedded Systems Handbook*. Taylor & Francis Group, LLC, 2006.
- [13] T. Chang, T. Yeh, and R. Miller. Gui testing using computer vision. In *The 28th international conference on human factors in computing systems*, 2010.
- [14] T. Cheng, K. Embedded software-based self-testing for soc design. In R. Zurawski, editor, *Embedded Systems Handbook*. Taylor & Francis Group, LLC, 2006.
- [15] J. Davis. Gme: The generic modeling environment. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '03*, pages 82–83, New York, NY, USA, 2003. ACM. ISBN 1-58113-751-6. URL <http://doi.acm.org/10.1145/949344.949360>.
- [16] S. Daya. Understanding statistics - the t-test for comparing means of two groups of unequal size. *Evidence-based Obstetrics & Gynecology*, 5(2):60–61, 2003.
- [17] S. Daya. Understanding statistics: Paired t-test. *Evidence-based Obstetrics & Gynecology*, 5(3):105–106, 2003.
- [18] eggPlant. eggplant functional: Automated mobile & desktop ui testing, 2015. URL <http://www.testplant.com/eggplant/testing-tools/eggplant-developer/>. Access date: 13 December 2015.
- [19] M. Emerson. A survey of embedded systems tools. In *ACM SIGBED Review - Special issue: Model-based design*, 2004.
- [20] A. B. G. Document analysis as a qualitative research method. *Qualitative Research Journal*, 9(2):27–40, 2009. doi: 10.3316/QRJ0902027.
- [21] T. Graves, M. Harrold, J. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 2001.

- [22] M. Grechanik, Q. Xie, and C. F. Creating gui testing tools using accessibility technologies. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 243–250, April 2009.
- [23] A. Henzinger, T., P.-H. H., and H. Wong-Toi. Hytech: the next generation. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 56–65, Dec 1995.
- [24] A. Henzinger, T., B. Horowitz, and M. Kirsch, C. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, Jan 2003. ISSN 0018-9219.
- [25] HTMLTestRunner. Htmlltestrunner - tungwaiyip’s software, 2014. URL <http://tungwaiyip.info/software/HTMLTestRunner.html>. Access date: 6 December 2014.
- [26] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.
- [27] JAutomate. Features | jautomate, 2015. URL <http://jautomate.com/features/>. Access date: 13 December 2015.
- [28] N. Juristo and A. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, 2001.
- [29] P. Karmore, S. and R. Mabajan, A. Universal methodology for embedded system testing. In *Computer Science Education (ICCSE), 2013 8th International Conference on*, pages 567–572, April 2013.
- [30] M. Kovacevic, B. Kovacevic, V. Pekovic, and D. Stefanovic. Framework for automatic testing of set-top boxes. In *Telecommunications Forum Telfor (TELFOR), 2014 22nd*, pages 1091–1094, Nov 2014.
- [31] V. Legourski, C. Trödhandl, and B. Weiss. A system for automatic testing of embedded software in undergraduate study exercises. *SIGBED Rev.*, 2(4):48–55, Oct. 2005. ISSN 1551-3688.
- [32] P. Li, T. Huynh, M. Reformat, and J. Miller. A practical approach to testing gui systems. *Empirical Software Engineering*, 12(4):331–357, 08 2007.
- [33] B. Meyer, I. Ciupa, A. Leitner, and L. Ling Liu. Automatic testing of object-oriented software. In *Theory and Practice of Computer Science*, January 2007.
- [34] M. Ould and C. Unwin. *Testing in software development*. Cambridge University Press, 1986.

- [35] Python Template Strings. 7.1. string – common string operations – python 2.7.11 documentation, 2016. URL <https://docs.python.org/2/library/string.html#template-strings>. Access date: 30 March 2016.
- [36] M. Qian, H. and C. Zheng. A embedded software testing process model. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1–5. IEEE, 2009.
- [37] Robot Framework. Robot framework, 2015. URL <http://robotframework.org/>. Access date: 10 February 2015.
- [38] D. Sale. *Testing Python: applying unit testing, TDD, BDD, and acceptance testing*. Wiley, Chichester, England, 1 edition, 2014;2013;. ISBN 1118901223;9781118901250;9781118901229;1118901258;9781118901243;111890124X.
- [39] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.
- [40] Sikuli. Sikuli script - home, 2014. URL <http://www.sikuli.org>. Access date: 16 August 2014.
- [41] E. Sjösten-Andersson and L. Pareto. Costs and benefits of structure-aware capture/replay tools. *SERPS'06*, page 3, 2006.
- [42] J. Tian. *Software quality engineering: Testing, quality assurance, and quantifiable improvement*. Wiley, Hoboken, N.J, 2005. ISBN 0471713457;9780471713456.
- [43] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, 2012.
- [44] R. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009. ISBN 9781412960991.
- [45] J. Zainzinger, H. Testing embedded systems by using a c++ script interpreter. In *Test Symposium, 2002.(ATS'02). Proceedings of the 11th Asian*, pages 380–385. IEEE, 2002.