Independent degree project – first cycle
*Självständigt arbete på grundnivå*

Industriel Engineering and Management
Industriell organisation och ekonomi

Automated GUI regression tests via remote desktop with Robot Framework and Sikuli

Michelle Ludovici
Automated GUI regression tests via remote desktop with Robot Framework and Sikuli

Author: Michelle LUDOVICI  
michelleludovici@gmail.com 
Civilingenjör i 
Industriell Ekonomi

Examinator: Aron LARSSON
Supervisor: Johanna SEFYRIN

Maria MORITZ

June 14, 2014
Abstract

To test software over a remote desktop is an increasingly common practice since outsourcing and offshoring of testing activities are on the rise. Most companies use proprietary testing tools which require an agent to be installed on the remote machine. This installation opens several attack surfaces and has thus the potential to compromise security. In addition license, support and other proprietary test tool costs might reduce the return on investment of automated tests to an unprofitable level, especially for relatively short projects.

Robot Framework with Sikuli integration (RFSI) is a concept based on two open source tools which offer solutions concerning the security and cost issues. RFSI based tests do not need any further installations, access rights or network configurations on the remote machine, because of the fact that the tests are based on visual input. Therefore tests with RFSI cannot access the AUT (Application Under Test) or the remote machine in any other way than a manual tester could. Open-source tools, being freely available, also escape the problem of licence costs. The purpose of the proof of concept pilot study was to give an answer on the question if RFSI can furthermore be considered an alternative to proprietary tools in terms of requirements for tool quality characteristics and for the external context of test automation projects.

The methodology of this research followed the broader Goal-Question-Metrics method (GQM) described by Rini van Solingen and Egon Berghout, based on the GQM approach by Victor Basili and David Weiss. The requirements and metrics for the evaluation of RFSI in the study were selected through a literature study of ISO/IEC-TR9126-3 and other articles in the field. They comprehend six categories: functionality, reliability, maintainability and stability, efficiency, understandability and portability. For each category, metrics were established and adapted to the context of the proof of concept. The scope of the project was given by six use cases and test specifications, established in cooperation with system experts at the company CGI Sundsvall. These were then translated into RFSI test scripts and implemented with the requirements for successful remote GUI testing in mind.

The results of the measurements for the proof of concept study showed that RFSI fulfills the internal and external requirements for successful automated testing completely for functionality, understandability and portability. The requirements are partially satisfied for reliability, maintainability and stability, and efficiency.

In conclusion, RFSI satisfies enough requirements to be considered a viable alternative to commercial testing tools for tests of GUI (Graphical User Interface) desktop applications over a remote desktop.
Acknowledgements

I would like to thank Johanna Seifyrin from Midsweden University, Maria Moritz and Johan Moritz from CGI Sundsvall for their valuable and constructive suggestions and expertise during the planning and development of this research work. Their willingness to give their time so generously has been very much appreciated.

I would also like to extend my gratitude to Christina Nilsson, Tomas Olsson och Ove Quist at SDC for introducing me to Robot Framework and thus inspiring me for the idea to this thesis.
# Contents

1 Introduction .................................................. 5  
1.1 Background ...................................................... 5  
1.2 Problem formulation ........................................... 6  
1.3 Purpose and aims ............................................. 7  
1.4 Scope and delimitation ......................................... 8  

2 CGI ................................................................. 9  

3 Theory and previous research ................................. 10  
3.1 The complexity of software testing .......................... 10  
3.2 Regression testing ............................................ 10  
3.3 Test specification, use cases, test cases, test scripts and test steps ............................................ 11  
3.4 Challenges of automated testing with tools ................. 12  
3.5 Automated GUI testing ......................................... 12  
3.6 Testing Tools .................................................. 13  
3.6.1 Robot Framework ...................................... 13  
3.6.2 Sikuli ...................................................... 13  
3.7 ISO/IEC-TR9126-3 ............................................. 14  
3.8 Requirements ................................................ 14  
3.8.1 Functionality requirements ............................... 15  
3.8.2 Reliability requirements ................................. 16  
3.8.3 Maintainability and stability requirements ........... 16  
3.8.4 Efficiency requirements ................................. 17  
3.8.5 Understandability requirements ....................... 18  
3.8.6 Portability requirements ................................. 19  

4 Research method ................................................. 20  
4.1 Planning phase ................................................ 20  
4.2 Definition phase ............................................. 21  
4.3 Data collection phase ......................................... 23  
4.4 Interpretation phase .......................................... 23  
4.5 Company support ............................................. 23  

5 Implementation .................................................. 24  
5.1 Use case 1 ..................................................... 24  
5.2 Use Case 2 ..................................................... 27  
5.3 Use Case 3 ..................................................... 28  
5.4 Use Case 4 ..................................................... 28  
5.5 Use Case 5 and Use Case 6 .................................. 31  
5.6 CleanUp ....................................................... 31  
5.7 Observations during implementation ....................... 31
## 6 Results

6.1 Functionality requirements ............................................. 32
6.2 Reliability requirements .................................................. 33
6.3 Maintainability and stability requirements ............................ 33
6.4 Efficiency requirements .................................................... 35
6.5 Understandability requirements ........................................ 35
6.6 Portability requirements ................................................... 36

## 7 Discussion

7.1 Metric selection ............................................................ 37
7.2 Degree of requirement fulfillment ...................................... 37
7.3 Thesis answers .............................................................. 38
7.4 Test tool recommendation ............................................... 38
7.5 Ethical and social aspects ................................................ 39
7.6 Future Work ................................................................. 39

### Appendices

A Robot Framework and Sikuli IDE ........................................ 46
B Requirement metrics ....................................................... 47
C GQM model for proof of concept ......................................... 57
D Use cases and test specifications ........................................ 58
D.1 Use case 1: Create, built, execute and delete a DecideIT decision tree . . 58
D.2 Test specification for use case 1 ........................................ 59
D.3 Use case 2: Build, add and execute additional decision trees .......... 62
D.4 Test specification for use case 2 ........................................ 63
D.5 Questionnaire ............................................................... 65
D.6 Test specification for use case 3 ........................................ 67
D.7 Use case 4: Create, count and pass SQL-selection to DecideIT tree for calculations ......................................................... 72
D.8 Test specification for use case 4 ........................................ 73
D.9 Use case 5: Choose email template, give layout and test-send email . . 74
D.10 Test specification for use case 5 ........................................ 75
D.11 Use case 6: Verify report in Report Portal ............................ 77
D.12 Test specification for use case 6 ........................................ 78
E Code from the library 'myCommandsMiluPC' and 'commonFunctions' 79
F RIDE reports and logs ....................................................... 81
G Test Statistics ................................................................. 83
H RIDE menu and BuiltIn library .......................................... 84
# Terminology

**AUT**  Application under test; the anonymous name of the GUI application which is tested remotely

**AUT project**  Application under test project; the anonymous name of the software project concerning the AUT.

**API**  Rules, protocols and tools to lay out communication between software components

**Build tool**  Programs that compile, link and package source code for execution

**Error**  The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

**Failure**  The inability of a system or component to perform its required functions within specified performance requirements.

**Function**  A function in this thesis is any piece of code that is invoked multiple times. Bundled automated test steps and whole test scripts can be considered functions if they are invoked more than once.

**GUI**  Graphical User Interface

**Open source definition**  When a software program is open source, it means the program’s source code is freely available to the public. Unlike commercial software, open source programs can be modified and distributed by anyone and are often developed as a community rather than by a single organization. The terms of use are often defined by the GNU General Public License, which serves as the software license agreement (SLA) for many open source programs. The distribution terms of open-source software must also comply with 10 criteria defined by the open source community [Open Source Initiative, 2014]

**Patch**  A modification made to a source program as a last-minute fix or afterthought.

**Proof of concept**  A proof of Concept is the delivery of a working system to prove that the technology works and functions as intended. It is usually small and may or may not include all functionality; instead, it refers to a partial solution that involves a relatively small number of users. Once complete, the PoC should be dismantled and considered complete after the results have been documented [Government of Newfoundland & Labrador, 2013]
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>To issue for performance, sale, publication, or distribution.</td>
</tr>
<tr>
<td>RIDE</td>
<td>The Robot Framework user interface in which test cases can be structured, libraries can be added and which displays the test reports and test logs.</td>
</tr>
<tr>
<td>RFSI</td>
<td>Robot Framework with Sikuli integration; an acronym stating that the implementation part of the project was done with both Robot Framework and Sikuli, taking advantage of both tools qualities.</td>
</tr>
<tr>
<td>Test case</td>
<td>A test case is a logical aggregation of actions/test steps belonging to a use case. Test cases describe actions between a user and a system more in detail than use cases. In the thesis each test case has been broken down to be transformed into a test script.</td>
</tr>
<tr>
<td>Test script</td>
<td>In the thesis, each test script contains the code for an automated test case. The terms test script and module are used synonymously during the thesis.</td>
</tr>
<tr>
<td>Test script stabilization</td>
<td>The effort of a tester in test script development to run, fail-check and improve test scripts progressively in order to reduce false negatives.</td>
</tr>
<tr>
<td>Test specifications</td>
<td>Test specifications specify the inputs, predicted results and execution conditions of tests. With the test specifications, the tester sees if the actual results of the test match the predicted results and can thus determine if a test has passed or not.</td>
</tr>
<tr>
<td>Test suite</td>
<td>To execute a test run of all test scripts combined is called a test suite.</td>
</tr>
<tr>
<td>Test steps</td>
<td>A test step is one unit of action which a user undertakes when interacting with the system. Test cases are built of test steps.</td>
</tr>
<tr>
<td>Use case</td>
<td>A use case is a high level description of steps that specifies how a user interacts with the business or system while keeping in mind the value that this interaction provides to the user or other stakeholders. In this thesis a use case has besides its high level description, a list of test cases and test steps, describing the use case more in detail. A use case is always a flow of several test cases running after each other, i.e. within a use case, the second test case can only run if the first test case has gone through.</td>
</tr>
<tr>
<td>Unit testing</td>
<td>The type of testing where a developer proves that a code module (the &quot;unit&quot;) meets its requirements.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

The Cold War Missile Crises September 26, 1983 could have ended in a full war, had it not been for Officer Stanislav Petrov who decided not to respond to an early warning satellite system which alerted Moscow about an US intercontinental ballistic missile launch. He made the right decision. An analysis showed later that the detection software misinterpreted reflections from the sun as missile launches and triggered the alarm. [Fordon, 2001] If there ever was a reason for thorough software testing this is one. No software is perfect from the beginning and besides preventing lethal fails in safety-critical (life-saving) software, testing can also save companies money. Research shows that costs for a software project increase the more, the later bugs are detected in the software [Michael et al., 2005; Newman, 2002]. A third reason for testing is that testing serves for verification and validation of contract requirements in software functionalities and quality. To find out that the requirements are not met when the product is in production, can be costly as Toyota found out as late as February 2014, when it had to recall 1.9 million vehicles globally to fix a software glitch that could shut down the vehicles automatically [Greimel, 2014].

To test software over a remote desktop is an increasingly common practice since outsourcing and offshoring of testing activities to countries like India or China are on the rise. Research by Nasscom confirms that: “Over the last 5-6 years, India has become one of the leading destinations for outsourcing of Software Testing Services […] By 2020, India’s Testing segment is expected to be USD 13-15 billion […] “ [NASSCOM, 2011]. The Wall Street Journal reports a study by TechNavio in 2013 according to which the “Global Outsourced Software Testing Services market is estimated to grow at a CAGR (compound annual growth rate) of 10.12 percent over the period 2012-2016” [The Wall Street Journal, 2013]. It is therefore important to address two drawbacks which become apparent when testing over the remote desktop with proprietary testing tools. The first drawback is that the usage of commercial tool based tests on remote machines allows for security breaches. Proprietary tools, such as QTP, IBM Rational Functional Tester, TestComplete and SilkTest require the testing tool or a software agent to be installed at the remote machine in order to “hook” into the AUT [Hewlett-Packards, 2009; IBM, 2010; Micro Focus, 2012; Smart Bear Software, 2014]. This necessitates network configurations on the remote machine in order to allow network access to the remote agent. For example QTP needs Distributed COM service configurations which were a point of attack

---

1To test over a remote desktop is necessary when the AUT cannot be set up locally. This is often the case when desktop applications are introduced, customized or maintained for servers in different countries, when a software company outsources testing or all at the same time
CHAPTER 1. INTRODUCTION

For the “Blaster Worm” in 2003 [CERT, 2003; Microsoft, 2003]. Of course access to the remote desktop service needs to be restricted, but a remote agent would for example be capable to detect and use hidden administrative features of an AUT that are normally not accessible to a restricted user. The agent would thus become a vulnerability and open up attack surfaces by allowing to circumvent user restrictions. Possible consequences can be injection and execution of arbitrary code in and from the AUT, gathering of sensitive information, gain of elevated privileges, scripting attacks, etc. [Kirn et al., 2000]. Another security risk example is that those agents may use OpenSSL for workbench-to-agent communication and are therefore vulnerable to the recent “Heartbleed” exploit [Avanthi and Stanley, 2013; Mardesich, 2014]. Naturally, security issues are handled through contracts that stress protection against security- and property right breaches. Nevertheless, a preemptive solution is desirable, especially when safety-critical, possibly compromising, expensive software, and the competitive advantage of innovation are at stake. It is the more surprising that research today seems not to have addressed the implications for automated tests over a remote desktop connection yet.

The second drawback with automated remote tests based on proprietary testing tools is the trade-off between costs, time and scope of software testing. Usually, those three factors cannot be optimized at the same time and companies find themselves confronted with a trade-off between them. For remote tests with commercial tools, this predicament is significant: Remote tests are automated for large software applications with the purpose of achieving a high ROI while delivering a quality application. But proprietary software is sold with expensive licenses which are often bound to a long contract period. If the AUT is on remote servers in different countries, the testing company needs to buy licenses for all remote machines in order to be allowed to install the tool/agent there. The costs of those licenses might reduce the ROI of automated tests to an unprofitable level, especially if the license contract period is longer than the testing period. On the contrary, if the testing company does not invest in a commercial testing tool, but does manual testing instead, the scope of testing will be much smaller within the same time frame, which reflects in the quality of the application. If the tests are done thoroughly, they will stretch the time frame.

1.2 Problem formulation

The question of how to best automate regression tests for GUIs over the remote desktop proposes a new challenge that has not been given much attention in research. Especially GUI tests over the remote desktop with other than proprietary testing tools has not been a topic in research. However, open-source tools could solve two drawbacks which the use of proprietary tools entails. The most common solution for proprietary tools when testing remotely is to install a software agent on the remote machine, but this procedure allows for security breaches (see chapter 1.1). Another inconvenience is that users may encounter a cost-time-scope trade-off, because of the license and support costs that come

2This problem can be illustrated with the famous software project triangle, also known as the “iron triangle.” [Atkinson, 1999]

3A seat license for QTP is tied to a certain computer and longer contract period, whereas a concurrent license allows any one person in a local network to use QTP. The concurrent license is just available for 180 days and the costs are not publicly available. Fees for for a seat license with the testing tool QTP were about $8000 - $12000 in 2012. [Jain, 2014]
with proprietary tools.

RFSI (Robot Framework with Sikuli integration) is one of the few open-source tools which is applicable for testing GUIs over the remote desktop. If RFSI can execute successful automated tests remotely, this could indicate a solution to the two drawbacks of proprietary testing tools. With Sikuli, the remote tester does not need any further installations, access rights or network configurations on the remote machine, because of the fact that the tests are based on visual input. Therefore tests with Sikuli cannot access the AUT or the remote machine in any other way than a human tester could. Open-source tools might also be a solution to the trade-off between costs, time and scope since they have the advantage of being free, while offering functionality comparable to commercial testing tools. It is left to prove if Robot Framework and Sikuli based tests satisfy the requirements for quality characteristics of software testing tools and for the external context of test automation projects, such that successful automated testing over a remote desktop is possible.

1.3 Purpose and aims

The goal of this proof of concept pilot study is to evaluate if Robot Framework with Sikuli integration (RFSI) is a viable alternative to proprietary testing tools in test automation projects over a remote desktop. To this end, requirements and metrics for the quality characteristics of a software testing tool and for the external context of test automation projects are defined. The thesis then assesses to which degree RFSI satisfies the presented requirements.

The thesis attempts to answer the following questions:

- What are academically acknowledged and common requirements for the quality characteristics of a software testing tool and for the external context of test automation projects?
- To which degree does the tool RFSI and RFSI-based test scripts fulfill the presented requirements in tests of GUI desktop applications over a remote desktop?
- Can RFSI be considered an alternative to proprietary tools for testing GUI desktop applications over a remote desktop?
CHAPTER 1. INTRODUCTION

In order to achieve the goal, the following artifacts will be delivered:

- Use cases and test specifications for the AUT, selected and approved in cooperation with a system expert on the project and an experienced test leader.
- Robot Framework and Sikuli based test scripts
- Requirement metrics and results
- Visualization of collected data

1.4 Scope and delimitation

The thesis covers regression testing at the system level and does not consider unit testing or integration testing. The thesis focuses on testing the desktop Windows GUI application, called “AUT” during the thesis, which means that functionality which is not part of the AUT, but could be found in other Windows applications are exempt from the research. The tests are written for selected use cases which have been identified by experts to be most central to the AUT project. The tests focus on the functionality of the AUT, not on the design or layout.
Chapter 2

CGI

The thesis research has been done in cooperation with CGI. CGI is the acronym for “Conseillers en gestion et informatique” and has been founded 1976 in Québec, Canada. CGI expresses both an internal and an external vision. Their internal vision is called their “dream”: “To create an environment in which we enjoy working together and, as owners, contribute to building a company we can be proud of” [CGI Group Inc., 2013]. The essence of CGI to the outside world is expressed as follows: “To be a global world class information technology and business process services leader helping our clients succeed” [CGI Group Inc., 2013]. CGI has a strong company culture which remained stable even through the acquisition of Stanley Inc. in 2010 and the Anglo-Dutch business and technology services company Logica in 2012. Today CGI is present in 40 countries, employs about 68 000 professionals and offers IT-services in several branches including Health, Government, Manufacturing, Finances, Logistics etc. In Sweden, CGI is represented in 35 cities, with their main office in Stockholm and about 5000 people were employed in Sweden in 2013 [Djurberg, 2013].

Up to date, CGI has been using QTP for testing purposes and is now interested to investigate alternative testing tools for a GUI application called AUT for a number of reasons: First of all the CGI plans to use Citrix clients in the future, since these clients are a complete solution of SSL, VPN and a custom remote desktop client [Citrix System Inc., 2013], which provides centralized authentication, confidentiality, encryption, second generation firewalls and network layer protection. The downside is that remote tests are problematic to accomplish with QTP on a Citrix client, because the QTP version has to match a certain Citrix version [Hewlett-Packard, 2014], both QTP and the AUT have to be installed within the Citrix server [QTP, 2008] and existing frameworks (for data storage for example) have to be available on the Citrix client. Another reason is that CGI is conscious about the fact that open-source tools can challenge established tools like QTP, since the open-source community is expediting development of modern and innovative software rapidly. CGI would like to capture this experience and knowledge by being in the frontline of successful innovations. In addition, if open-source tools have capabilities comparable to QTP, CGI would be able to augment their repertoire of automated tests with open-source tools to the advantage of cutting down fees for smaller projects which do not necessitate the all-round features that QTP is offering. Clients might appreciate this choice which competitors might not be able to offer.

---

4 which customers may not agree too
Chapter 3

Theory and previous research

In this chapter, first two drawbacks are mentioned concerning remote tests with proprietary testing tools. Then, the complexity of software testing is described, an overview on what regression testing means is given and test specification, use cases, test scripts and test steps are explained. It follows an overview over the challenges of automated testing with tools, an explanation on what automated GUI testing is about and an overview over the testing tools used in the proof of concept. Finally the theory chapter explains the international standard ISO/IEC-TR9126-3 with its metrics for the measurement of external quality characteristics of software products and finishes with theory about requirements for successful automated GUI regression testing.

3.1 The complexity of software testing

Testing is a process of verifying and validating that a software application or program
1. Meets the business and technical requirements that guided its design and development, and
2. Works as expected.
Software testing also identifies important defects, flaws, or errors in the application code that must be fixed. [Bentley, 2005]

Software Testing has developed into a complex field of study with its own glossary, specializations and best practices. Padmini [2004] has written an overview of testing terminology. According to him, software can be tested at three basic levels: unit testing (tests code of a program), integration testing (verifies the interaction between components) and system testing (tests the behavior of the entire system against the requirements). Different testing types are distinguished which describe how tests are performed, by whom and how the system to test is viewed (manual-, automated-, formal-, informal-, black box-, white box testing). Testing is also distinguished by intended purposes (functional-, load-, performance-, end-to-end-, sanity-, regression-, acceptance testing, etc.). Related to testing, there are different testing frameworks which are categorized according to their method of operation (keyword driven, linear, hybrid, data driven, etc.) [Padmini, 2004].

3.2 Regression testing

Regression testing at the system level is necessary in order to verify that a software system under modification does not loose functionality when new features are added or
existing features are reshaped. This is relevant because software requirements change
frequently as programs are built in relative short phases of development and then re-
ceive additional features, are adapted for other purposes, or transition into a long phase
of maintenance and support during which users report system errors. While parts of a
program are being revised, regression testing insures that the rest of the program works
as intended [Lin, 2005]. Large software systems may pose a challenge for manual testers
since test cases and test suites grow rapidly with additional program features while each
regression test cycle still has to cover all existing test cases and test suites. When the
system has a lot of functionality to test and when a program has a long product life cycle,
automated regression testing can be an essential aid in reducing the increasing manual
testing costs and labor, the overall testing time, increasing the coverage of test steps, im-
proving visibility and insuring repeatability of tests. Nevertheless, automated regression
testing is not a panacea, nor is it always worthwhile as the next sections of the thesis will
describe.

3.3 Test specification, use cases, test cases, test scripts
and test steps

Test specifications specify the inputs, predicted results and execution conditions of
tests [IEEE, 1990]. With the test specifications, the tester sees if the actual results of
the test match the predicted results and can thus determine if a test has passed or not.
Test specifications contain use cases, test cases and test steps. A use case is a high
level description of interactions between a user and the business or system which creates
value [Heumann, 2008]. In this thesis a use case has besides its high level description,
a list of test cases and test steps, describing the use case more in detail. A use case is
often a flow of several test cases running after each other. The test cases themselves are
independent scripts. "A test case is a set of test inputs, execution conditions, and expected
results developed for a particular objective, such as to exercise a particular program path
or to verify compliance with a specific requirement" [IEEE, 1990]. Test cases contain
test steps, while test scripts contain the code for automated test cases. A test step is a
description of the smallest unit of action which a user undertakes when interacting with
the system. The terms test script and module are used synonymously during the thesis.
To execute a test run of all test scripts combined is called a test suite. In a nutshell, use
cases contain test cases, which are built of test steps.
CHAPTER 3. THEORY AND PREVIOUS RESEARCH

Figure 3.1: Test Specification example

All test steps that are marked green represent alternative flows, i.e. exceptions or alternatives that could occur during execution and therefore have to be incorporated into the test specifications.

3.4 Challenges of automated testing with tools

With the rise of larger and more comprehensive software systems, testing and especially regression testing has become a challenge which is tackled by automation. Automated testing is software testing that is being automated with the help of a software tool or by programs\(^5\) [Hayes, 1995]. The tool or program will execute the test scripts automatically, which means the tester does not have to be involved after playing the test. Most testing tools for test automation have capture-replay functionality, such as QTP or Selenium. Those capture-replay tools record a tester’s action (for example clicking on a button) and are then able to replay the action. The use of capture-replay alone has serious limitations to script writing since such tests will be restricted by the features of the testing tool, since simple capture-replay tests scripts are static [Damm et al., 2005]. In static scripts, the use of dynamic elements like variables or adding new code is not possible. A login test-script could for instance just log-in with one user and his/her password, but the tester would have to write several similar test scripts in order to try out different passwords. Fortunately the limits of most test tools can be escaped with a basic knowledge of coding because automated test tools often include API’s (application programming interface) or can be extended with API’s which allow for writing dynamic scripts. Dynamic scripts have advantages, such as allowing for the use of variables, highlighting dependencies in the code or reusing code. In chapter 3.8 which describes the test tool and other requirements for successful automated testing, it is implied that test scripts are dynamic. The question of how to best automate regression tests for GUIs proposes a new challenge that has not been given much attention in research until recently. The pool of commercial testing tools and open source testing tools that are available today are mostly equipped to test web- or mobile applications and encounter a variety of problems when being used for testing GUI applications, not to mention testing GUIs over a remote desktop.

---

\(^5\)Writing programs for testing is mostly done in unit testing. Unit testing is not a topic in this thesis.
3.5 Automated GUI testing

“GUIs today constitute as much as 45-60 per cent of total software code” [Memon et al., 2005]. The challenges in automated testing described in the chapter above are in general valid for all kinds of automated tests, but automated regression testing for GUIs is even more difficult. Researchers frequently emphasize the challenges of regression testing of GUI applications. Pettichord B. underscores that “Major challenges for GUI test automation are maintainability and reliability” [Pettichord, 1996]. Memon and Soffa [2003] also address the problem of unusable test cases and “expensive regeneration” because of changes in the layout of the GUI. An example given is GUI test scripts that break when one element (for example a button) is moved to another menu or has changed its color. Research by Ames and Jie [2004], and by Memon et al. [1999] expound the problem of the large amount of possible interaction paths with a GUI:

[...] the space of possible interactions with a GUI is enormous, in that each sequence of GUI commands can result in a different state, and a GUI command may need to be evaluated in all of these states. This results in a large number of input permutations [...] what matters is not only how much of the code is tested, but in how many different possible states of the system each piece of code is tested. An important aspect of GUI testing is verification of its state at each step of test case execution”. [Memon et al., 1999]

The problem of many testable interaction possibilities exposes the necessity to prioritize test cases. The problematic is furthermore complicated by rapid, agile changes in software development that accompanies each new patch, upgrading or release. A preliminary literature search about the subject of remote automated GUI tests over the remote desktop revealed that literature in this specific area is scarce at the time of writing.

3.6 Testing Tools

In the following section the two open-source tools Robot Framework and Sikuli are presented.

3.6.1 Robot Framework

“Robot Framework is simple, yet powerful and easily extensible tool which utilizes the keyword driven testing approach” [Stresnjak and Hocenski, 2011]. Robot Framework is an open-source tool based on Python which makes it compatible with all major platforms. It uses existing libraries to run actions on the system under test and can easily “as-similate” new libraries (written in Python or Java) to extend its functionalities. Keywords from those libraries are used to write test cases in a table format and new keywords/libraries can be created by the user. Robot Framework has a modular architecture. This entails separation of test data, the framework itself, the test libraries and the system under test, which has the advantage that a change in one module will not affect the remaining parts. Robot Framework comes with a GUI called RIDE (Robot Framework Integrated Development Environment), which can be seen in appendix A. RIDE facilitates editing
and adding test cases, test suites, libraries and variables. The tabular format for entering keywords and arguments offers visual help by greying out areas when no further arguments can be entered or reddening areas with wrong keywords or arguments [Stresnjak, 2011]. Furthermore Robot Framework generates test reports in HTML-format which color-mark failed and passed test cases. A more detailed log report shows the appropriate error messages, the execution time for each test step, the test suite, test case, test step, table and the variable (if applicable) for which the error occurred. Robot Framework is maintained regularly [Klärk et.al., 2014], sponsored by Nokia Siemens Networks and released under Apache Licence 2.0.

3.6.2 Sikuli

Sikuli uses image recognition to automate anything that can be seen on the screen and does therefore not need a GUI's internal code. For the practical implementations of tests, the SikuliX version of Sikuli is used, which is maintained and developed by Raimund Hocke and the open-source community [Sikuli Doc Team, 2012]. Sikuli itself is programmed in Java, but uses Python-based scripts. The tool has an IDE (Integrated Development Environment) and matches predefined screenshots with the elements on the screen on which actions are to be performed. The test scripts can be written with visual elements (for example: 

```
click(Photo-Open)```
)

or directly be programmed in Python. The screenshots for a certain test case are all stored in the same folder. Sikuli uses the Python library for functions and offers Java API [Sikuli Doc Team, 2012]. A screenshot of the Sikuli IDE can be seen in appendix A.

3.7 ISO/IEC-TR9126-3

"ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) form the specialized system for world-wide standardization" [ISOIEC9126-3, 2002]. The technical report ISO/IEC-TR9126-3 gathers internal metrics for the measurement of six internal quality characteristics of software products, presented in figure 3.2. Developers, evaluators, quality managers and acquirers may select metrics from this technical report for defining requirements, evaluating software products, measuring quality aspects and other purposes. They may also modify the metrics or use metrics which are not included here. This report is applicable to any kind of software product, although each of the metrics is not always applicable to every kind of software product" [ISOIEC9126-3, 2002]. For the proof of concept the ISO/IEC metrics have been adapted to measure the degree of fulfillment of the requirements presented in chapter 3.8 for the evaluation of the testing tool RFSI.
CHAPTER 3. THEORY AND PREVIOUS RESEARCH

15

Figure 3.2: The ISO/IEC 9126 for software product quality [AlbenFaris Inc., 2014]

3.8 Requirements for successful automated GUI regression testing with a testing tool

Automated GUI regression tests have to fulfill certain requirements in order to be meaningful. Those requirements concern the internal quality characteristics of the testing tool, as well as external regression testing requirements such as the project duration and resources. This research for requirements for successful automated testing with a testing tool combines thus both internal software quality requirements found in ISO/IEC-TR9126-3 with a literature study of articles focusing on external requirements for automated regression tests. The requirements for successful automated GUI regression testing are presented below with a short explanation of the metrics. Detailed summary of purpose, application, formula, inputs and results for the metrics can be found in appendix B.

3.8.1 Functionality requirements

Functionality is "The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions" [ISO/IEC, 2000]. A testing tool should have enough functionality to cover crucial test steps of a test specification. A tool which is too limited to test what is important is obviously not worth implementing. Myers et.al. summarize this concept with "Ceiling": “[…] the “ceiling” is how much can be done using the system” [Myers et al., 2000]. The testing tool should have the capabilities to execute the test steps in the test specifications for the AUT. Ideally the testing tool should not possess too many functions either, since unnecessary functions may impede usability.

A first metric for functionality in the ISO/IEC is "functional adequacy", which measures the number of tool-inherent functions that are suitable for performing the specified test steps. Functional adequacy is measured by taking the ratio of the number of functions which showed limitations within the project context (inadequate functions) to the
total number of functions implemented. An example of an inadequate function would be a function for counting money, which only supports pounds, whereas the project context requires the Swedish currency. This ratio is subtracted from 1 to get the percentage of adequate functions (functional adequacy). [ISOIEC9126-3, 2002]

A second metric for functionality is "functional implementation". The functional implementation of RFSI measures the number of limitations encountered during the implementation of self-written functions in relation to the number of implemented functions, for the purpose of RFSI customization. Limitations refers to functionality requirements which could not be implemented because of constraints in RFSI integration. This measurement gives an idea about the limitations of customizing RFSI for the testers purposes.

A third measure for functionality is "functional implementation coverage". For this metric, the number of test steps that could not be covered in relation to the total number of test steps is counted. [ISOIEC9126-3, 2002]

Furthermore, the existence of functions that allow for starting and stopping the test execution, automatically log, trace back and report details of test failure, customization of test reports, the ability to run tests in the background\(^6\) and separation of test data and test scripts, are external functional metrics for the functionality of a suitable testing tool [Vishwa et al., 2013].

### 3.8.2 Reliability requirements

ISO/IEC 9126-1 defines reliability as "The capability of the software product to maintain a specified level of performance when used under specified conditions" [ISO/IEC, 2000].

The absolute worse thing that can happen to an automated testsuite is that it reports that tests have passed when there are in fact problems with the functions they should be testing. This situation is called a false positive. If your testsuites get a reputation for false positives, no one will want to use them. They’d rather do the testing manually." [Pettichord, 1996]

False positives are for example possible if the test scripts are written incorrectly, i.e. they test something else than what they are supposed to or if test steps from the specifications have been overlooked in the implementation. False positives can only be spotted by observation of test runs, validations of test steps and by double checking test steps with the test specifications. False negatives are part of writing automated scripts. They are often caused by unstable scripts during the script stabilization time, which is the time in which the tester runs, fail-checks and improves test scripts progressively. False negatives are spotted when a test run shows a fail, whereas there is nothing wrong with the application. An effort to include all test steps in automation, stabilization of test scripts and regular maintenance activity will keep false negatives at their minimum. Two metrics for reliability are thus the occurrences of false positives/negatives in relation to the total number of test runs. The fact that a testing tool can handle its own system errors and failures is a final measure of reliability [Vishwa et al., 2013].

\(^{6}\)allowing the tester to execute other operations in the meantime
3.8.3 Maintainability and stability requirements

Maintainability is "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications" and stability is "The capability of the software product to avoid unexpected effects from modifications of the software" [ISO/IEC, 2000]. As the system under test undergoes frequent changes until its release, it is necessary to maintain test scripts in order to adapt them to new patches, upgrading or new development. It is important that regression testing produces executable tests fast to give rapid feedback to programmers, but also because of cost considerations [Ruiz and Price, 2008]. Therefore testers should not spend much time maintaining existing scripts [Ruiz and Price, 2008].

Maintainability can be reduced by logging all test actions, so that the tester can easily trace an error back to the right script and right code line. A first measure of maintainability is therefore the number of test steps that are logged in comparison to the total number of test steps in the test specifications [ISOIEC9126-3, 2002]. This serves the purpose of facilitating the understanding of a flow of activities from a script by logging those activities to the test log.

A second measure of maintainability is the cyclomatic complexity (MCC) invented by McCabe [Frappier et al., 1994]. In graph theory, the cyclomatic complexity is the number of linearly independently paths in a graph. In software scripts, each logical AND/OR decision is splitting a linear path into two other pathways. In scripts where only binary decisions are made, f.ex. non-nested if-statements, the usage of a simpler formula of the McCabe cyclomatic complexity is allowed: $MCC = D + 1$, where $D$ is the number of decision elements, i.e. operators such as "if", "case" or "for". The MCC measures the difficulty of understanding a script, but is also used for defining script size. "Researches have showed [sic] that probability of the fault increases in modules with MCC>10" [Panchenko, 2006].

A third external measure for the stability of automated tests is the ability of a testing tool to create tests which allow for changes in the GUI design without breaking, since GUI changes are a usual occurrence during development of GUI desktop applications. When for example the color or position of a GUI element changes, the testing tool should be able to find the element nonetheless and execute the tests without much maintenance [Ruiz and Price, 2008]. The fact that tests may or may not break with incidents such as element color-, position- and size-changes is another metric for maintainability and stability.

A fourth measure for stability of tests is the defect trend measure. This metric measures the number of known defects per number of test procedures executed [Garrett, 2012]. In the proof of concept, the number of false negatives (defects in the test scripts) are counted in relation to the total number of test runs executed over a period of seven days (active test writing and stabilization period per use case). The defect trend measure is expected to reduce the longer the test period has gone, which indicates stabilization of test scripts.
3.8.4 Efficiency requirements

ISO/IEC 9126-1 defines efficiency as "The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions" [ISO/IEC, 2000]. Regression testing occurs often under a high pressure period before the release of a product in order to find and eliminate as many remaining defects of the program as possible. It is therefore essential that the tests cover as many functions of the program under test as possible in a relative short time, i.e. the testing has to be efficient and produce results fast [Memon, 2002]. Efficiency means to reduce the time-to-market for automated tests. To maximize efficiency is equal to reduce the time it takes for setting up the testing tool and the test environment, the time it takes for writing the test scripts and the time for test execution [Muthusundar, 2011]. A summary on articles which use amongst others a low execution time as requirement for test automation can be found in the article “A Uniform Representation of Hybrid Criteria for Regression Testing” [Memon et al., 2013].

Measurements of efficiency are highly project and individual-experience dependent and may therefore serve as a future reference for research in this domain or for test managers to estimate testing efforts for similar projects. The measurements for efficiency were taken for an inexperienced tester with an unknown tool for a newly encountered AUT project.

Measurements of efficiency could for example be the average number of test steps implemented per day and the average number of test runs needed until stabilization. From those measurements, the maximum number of test suite runs possible per day can be estimated as a third future reference.

ISTQB (International Standard Testing Qualifications Board) recommends that between 30% and 40% of total development time is to be spend on testing [Franco and Peeters, 2006]. The metrics for the total testing time (i.e. testing of the whole AUT) in relation to total development time can be extrapolated from measurement data and expert estimations to evaluate if the total test time for the AUT project was within 30-40% of total development time for the AUT.

3.8.5 Understandability requirements

Understandability is defined as "The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use" [ISO/IEC, 2000]. Regression tests can have different users. For example can programmers run ready-made scripts to verify if their code is stable, fellow testers and even the end user of the AUT can be users of test automation scripts by executing and maintaining scripts. The end user of the test is required to have basic knowledge of how automation works, but should not have to spend much time to decipher the code of the scripts, because of unclear naming or a chaotic structure.

Informal coding conventions should improve the readability of the software, al-
lowing engineers to understand new code more quickly and thoroughly" [Oracle, 1999]. Since automating tests involves programming, those conventions should also apply to writing test scripts. The tester should use unique and unambiguous names for variables, functions, keywords and libraries, which are stored in logical and easy to find places.

Emery points out that test functions and variables should describe the essence of test steps [Emery, 2009]. He gives an example where a function is named "Reject Password" and takes different variables as parameters which represent passwords. Emery points out that it is essential to name the different variables by their essence for ease of understanding, as for example: ${PasswordTooShort}, ${PasswordContainsForbiddenCharacters} or ${PasswordContainsJustNumbers}. The use of naming convention is a first metric for understandability.

Writing understandable code also involves setting execution traces for code parts, i.e. writing a comment for each action that is executed in the script, so that errors can easily be traced back to a certain action in a certain script. This tracing back is done frequently for debugging of test scripts. The number of traces in relation to the number of test steps is a second metric for understandability of scripts. The related metric "activity recording" is used for measuring maintainability [ISO/IEC9126-3, 2002]. The difference between execution traces and activity recording is that in the number of comments which can also be retrieved in a test log are counted in activity recording. It is expected to have more comments within a test script (execution traces) than comments passed to a test log (activity recording). This is due to the fact that it is vital to not only understand the flow, but also the meaning of all functions and decision elements for debugging.

A third metric for usability is the testing tool’s ability to write verifications for each successful test script pass. Verification is done at the end of each test script by writing "OK" to an external file after each successful test script execution. If a Sikuli test script would fail, the "OK" assertion would not be written to the file and Robot Framework consequentially marks that test script as "Fail: AssertionError" in the log.

A final external metric for usability is the fact that documentation, tutorials and examples are available which explain the testing tool and the test writing, implementation and executing process [ISO/IEC9126-3, 2002].

### 3.8.6 Portability requirements

Portability is "The capability of the software product to be transferred from one environment to another [...]" [ISO/IEC, 2000]. Portability is an external requirement for automating tests with a testing tool and metrics for portability are: platform and OS independence, the existence of additional API’s, add-ins or modules and compatibility with building tools (f.ex.: Maven or Ant)[Vishwa et al., 2013].

or established by big companies for internal use
Chapter 4

Research method

The research method chapter describes how the approach for the thesis followed the steps of the "Goal - Question - Metrics model" (GQM model) during all phases of the proof of concept.

The GQM idea was originally developed by Basili and Weiss, and matured into a broader model described by van Solingen and Berghout (see figure 4.1) [Van Solingen and Berghout, 1999].

Figure 4.1: The GQM model by Berghout and Solingen

4.1 Planning phase

During the planning phase an improvement area was described: open-source solutions for automated GUI regression tests over a remote desktop which are as adequate in project use as proprietary tools and in addition escape the security and cost-time-scope tradeoff. A preliminary literature search about the subject of remote automated testing revealed that this particular problem seemed to be a research gap. A thorough search on diverse databases, such as ACM Portal, IEEEExplore, JSTOR, Springer, Ebsco, and on the internet for enterprise white paper with combinations of the search keys “Remote”, “desktop”, “connection”, “access”, “RDP”, ”software”, “testing”, “test”, “automated”, ”GUI” and “regression” did not reveal any useful results concerning the problem under scrutiny. The next step of the planning phase was to select an application project: a
remotely accessible GUI desktop application\textsuperscript{10}, the AUT. This project served as platform for evaluating Robot Framework with Sikuli integration as a potential solution to the improvement area in a proof of concept pilot study. As the thesis investigates amongst other the possibility to replace proprietary test tools with an alternative tool in tests via remote desktop, the use of Sikuli or Robot Framework alone does not offer enough functionality. A system that is viewed over a remote desktop window is usually seen as just one big picture by testing tools – this is also the case for Robot Framework. The inherent Robot Framework capabilities for GUI desktop application testing had therefore to be enhanced in order to test over the remote desktop. This is done by integrating the visual testing tool Sikuli with Robot Framework. Sikuli is able to identify each element on the screen by visual input alone, which facilitates writing test scripts and allows for remote testing, whereas Robot Framework uses different libraries which facilitate running test cases, augment testing features and has the possibility of printing test reports, logs and statistics. The integration of Robot Framework and Sikuli makes powerful testing over the remote desktop possible and this concept is referred to by the acronym of RFSI (Robot Framework with Sikuli integration) in the thesis. This integration is based on the fact that Robot Framework can run and pass information to and from other programs as part of test-cases. The testing tools are thus able to collaborate and pass test-data and results to each other by means of Python functions. The details of how the integration is implemented are described in the implementation chapter.

The scope for the proof of concept was defined by a selection of use cases which are broken down into test cases with test steps. The first criteria for the selection of use cases was to choose a broad variation of tests that are possible to implement on the AUT, such as mixing web-and desktop application testing over remote desktop, SQL-statements, and tests involving browsing through documents or waiting for execution results. The second criteria was to cover the most important parts of the AUT. The use cases were chosen in cooperation with Johan Moritz, a system expert on the AUT. The test cases and test steps were then assembled into a test specification, the completeness, utility and validity of which was confirmed by Maria Moritz, an experienced tester and project leader at CGI.

4.2 Definition phase

The next phase of the GQM model is the definition phase in which the goals (conceptual), questions (operational) and metrics (quantitative) are being set. Since it is a given that security and costs are not an issue for RFSI (see chapter 2.1), the specific project goal is to evaluate if the functionality of RFSI is sufficient to successfully test all the required elements of the AUT in order to evaluate if RFSI is able to compete with proprietary tools in the domain of automated software testing via remote desktop. A literature study was ground for finding questions which would lead to an answer on how to evaluate RFSI. The literature study revealed the ISO/IEC-TR9126-3 standard to be a suitable framework for evaluating the testing tool’s internal quality characteristics, since this international standard was created for just that purpose. Nevertheless, metrics in ISO/IEC-TR9126-3 had to be adapted to the purpose of this thesis.

A first selection criteria for quality evaluation metrics of ISO/IEC-TR9126-3 elim-\textsuperscript{10}The planning phase encompasses more steps, such as selecting a project team. Since those steps of the GQM model do not apply for the thesis, they will be ignored
inate metrics irrelevant in the context of the GUI environment and the remote desktop environment. For example, metrics for memory utilization were not relevant, since RFSI was dominantly limited by the remote desktop communication time and the rather slow AUT.

A second selection criteria for metrics eliminated those metrics which required to follow the AUT project over a longer time and were therefore out of the scope set for the proof of concept. Usability measures were left out for this reason, which could otherwise have been measured with real test users in the acceptance phase of the AUT.

Other metrics were left out because they were generally not applicable to RFSI, as for example security metrics since the tools Robot Framework and Sikuli do not have any authentication or security restrictions.

In order to tailor the research of requirements to the domain of automated testing, the metrics from ISO/IEC-TR9126-3 are completed by articles focusing on external requirements for automated (regression) tests. Additional external requirements for successful regression tests were extracted from those articles and combined with requirements found from ISO/IEC-TR9126-3. The selection criteria was to adopt the requirements and metrics used in several articles, such as efficiency metrics: Test execution time is a metric that is presented in several research articles, over ten of which can be found in the research-summarizing article “A Uniform Representation of Hybrid Criteria for Regression Testing” [Memon et al., 2013].

The collected prerequisites for successful automated GUI regression testing crystallize into the following broader questions:

- Does RFSI have the capability to execute all test steps designed for the AUT?
- Is RFSI portable?
- Is RFSI efficient enough for successful automated testing?
- Is RFSI reliable?
- Is RFSI maintainable and stable?
- Is RFSI understandable?

Those questions have then been broken down into more detailed questions in order to translate them into defined metrics, which "provide all the quantitative information to answer the questions in a satisfactory way" [Van Solingen and Berghout, 1999]. The detailed questions with the corresponding metrics can be found in appendix B. Figure 4.2 gives an overview over the definition phase for the proof of concept.
4.3 Data collection phase

In the data collection phase of the GQM model, the test scripts are implemented. Test steps were translated into RFSI based test scripts, while adhering to the requirements of successful automated regression testing. After the implementation and collection of data, a final step of the data collection phase is to turn raw data into processed data and then into graphs and tables.

4.4 Interpretation phase

In the interpretation phase of the GQM model, the metrics measurements and observations are turned into answers to the corresponding questions, which finally determine if the goal has been achieved or not. During this phase, the results of the practical implementation are analyzed and discussed in order to evaluate to which degree the test scripts could be implemented with the requirements for successful automated regression testing in mind. The GQM model for the planning, definition, data collection and interpretation phases carried out in this thesis can be found in appendix C.

4.5 Company support

During the research, Maria Moritz was the coordinator, advisor and handler of all administrative and organizational issues. Johan Moritz, a member of the AUT project team chose and approved the use cases. Maria Moritz reviewed the test cases and test steps.
Chapter 5

Implementation

The implementation chapter illustrates how the core aspects of each use case have been implemented. In order to understand the context better in which the implementation happens, this chapter is supported and refers to use cases and test specifications in appendix D.

Although the research and test-scripts in this thesis have been made for the AUT project, all use cases, test steps, screenshots, illustrations and explanations for the practical implementation of tests that are covered in the thesis will convey examples from applications other than AUT for non-disclosure reasons. The use cases, test steps, examples and explanations chosen for illustration purposes are selected to match the functionality covered in the AUT project and the working principles behind writing the tests in RFSI (Robot Framework with Sikuli integration) are the same for the AUT project, implementation of scripts as for the examples given in the thesis.

5.1 Use case 1

Use case 1 allows the user, a decision analyst, to create an example decision tree in the program DecideIT. The decision tree can serve as a basis for the decision analyst to evaluate the decisions and risks associated with different events and possibilities. The use case starts by creating a basic decision tree with 2 nodes and then customizes the tree by adding an event node with probabilities and values. Use case 1 is found to be a good introduction to the basic functionalities of RFSI. Sikuli has most basic functionalities to imitate a normal user’s mouse movements. Examples of basic functionalities are: "click", "rightClick", "drag-and-drop", "scroll" and "type". A typical example of a Sikuli script with basic actions is displayed in figure 5.1. If an element cannot be found, Sikuli will retry until it reaches a pre-defined time-out, at which point Sikuli will notify about a test-step failure. The timeout time can be defined.

Figure 5.1: A Sikuli Script with basic functionalities

11For definitions of and relation between test specification, use case, test case and test steps, see terminology chapter
Besides the basic functions, Sikuli has also a function with which the speed of execution can be controlled to a certain degree by importing the Sikuli Java class "Vision":

```java
from org.sikuli.basics.proxies import Vision
Vision.setParameter("MinTargetSize", 6.0)
```

Sikuli has also the possibility to show actions as the script plays. This mode is enabled with the function: `setShowActions(True)`. In that case, the screen-location of the action to be performed is indicated with a bull’s eye before execution and the execution time will be slower. Sikuli actions are structured and logged within RIDE, the Robot Framework IDE. Figure 5.2 shows the structure of Use case 1, which can be found in appendix D.

In order to fulfill use case 1, it was successfully tested to pass parameters from RIDE to Sikuli. The tester can define an input in RIDE, for example the question: "Use Automation?" and the Sikuli script reads and uses this input during execution. Sikuli can thus write "Use Automation?" in the name field of a decision tree. If the user wants the tree to be called another name, he/she can change the input next time. In order to pass a parameter from RIDE to Sikuli, the input parameter "ThisISaParameter" in figure 5.2 is saved in an environment variable called "This_Is_an_Environment_Variable" with help of the Python function:

```python
def set_env(self, name, value):
    os.environ[name] = value
```

This environment variable is later retrieved from Sikuli with the command: `myCommandsMiluPC.get_env('This_Is_an_Environment_Variable')`. The parameter is returned by the Python function:

```python
def get_env(self, name): return os.environ[name]
```

![Figure 5.2: The Robot Framework interface provides structure to Sikuli scripts and logs actions to a report](image-url)
The functions of RFSI have been augmented by creating an own library with customized Python functions. All customized functions are saved in a library called "myCommandsMiluPC", which is imported into RIDE. For example, Robot Frameworks capacity to log events has been deemed insufficient when integrated with Sikuli, since Robot Framework logs only if a Sikuli test case has gone through or not, but not the steps that Sikuli performs in between. This would nevertheless be desirable in order to keep maintenance low. If a test script error occurs, the tester could check at the last logging where the script stopped and directly see why, and where to change the test script. Therefore a log message has been made, with which each single action can be commented in a Sikuli test script, each comment is subsequently written to a file, which is in turn incorporated into the log function of RIDE. The following example will illustrate this process. In the Sikuli script, the following function passes the comment "Click OK" (which describes a Sikuli action) to the "myCommandsMiluPC" Python library with the following call:

```python
myCommandsMiluPC.currently("Click OK")
```

The Python function which takes in the comments and passes them to a file is defined as follows:

```python
def currently(self, message):
    self.write_file("PathToFile/sikuli_currently_doing.txt", "\"<p\" + message + "\"<p\", "a")
```

The code from the library 'myCommandsMiluPC' can be found in appendix E. In RIDE, another function reads the file with the log messages and passes them to the RIDE logging function:

![Image](image_url)

**Figure 5.3:** RIDE commands to read actions from file and log them

A small part of the RIDE test execution log shows that the action has been logged successfully:

![Image](image_url)

**Figure 5.4:** RIDE Test Execution Log

More detailed images of how RIDE test reports and test logs look like can be found in appendix F. In use case1, the test case 2.0 "Build example decision tree" can always
be expanded by adding more decision trees to the first one. Use case 2 exemplifies how this can be done.

5.2 Use Case 2

Use case 2 allows the user, a decision analyst, to build and add an additional decision tree to one of the branches of the main tree created in use case 1 in the program DecideIT. If for example the main tree is present, it can be further augmented by building another tree and drag-dropping it to a branch of the main tree. Use case 2 describes the test steps which are necessary in order to recursively build and add a selection of those additional trees to the main tree and then execute calculations on the resulting tree.

Use case 2 and the corresponding test specifications in this example are fictional in order to exemplify the test steps for the AUT project. To execute the test steps as described in the specification for use case 2 is therefore not possible in the program DecideIT\textsuperscript{12}, but for example purposes of the functionalities tested in use case 2 of the AUT, let assume the test steps described in appendix D can be done. Let the decision tree from use case 1 be called the "main tree". In total, it was requested to test building and adding 5 of 23 possible different additional trees to the main tree and execute calculations on the resulting trees. To build, add and execute one additional tree comprises in average 34 test steps. Because of the resemblance in how those 5 additional trees were being built, part of the code in the test script of use case 2 could be reused multiple times. This was done by "bundling" repeating test actions into an isolated function which would execute those test steps whenever the function was called. This could reduce possible coding mistakes and the tester can save time by not coding the same sequence repeatedly. For example, the following repeating actions are bundled into a function called "ok_opActive_ok":

```
myCommands.currently("Click OK, Select Operation active and click OK again")

click{OK}

click{Operation active\(\text{}\)OK}

Figure 5.5: three commonly repeated steps and a log message for traceability
```

This function will execute the above four steps whenever it is called:

```
commonFunction.ok_opActive_ok(inside())
```

Bundling of repeating actions was done repeatedly during the implementation, not only for bundling test steps together, but also for bundling whole scripts. This was for example done in use case 2, test case 2 which executes calculations on previously built additional decision trees. The test scripts to execute calculations on additional decision trees involve exactly the same actions except for one step, where the user has to right-click on a decision tree image which is unique for each decision tree. In the test specifications for use case 2, test case 2 ("Execute Calculations with additional tree", see appendix D), this step is marked in light blue. The steps of the scripts for use case 2, test case 2, were

\textsuperscript{12}it is though possible to augment an existing tree with additional branches
5.3 Use Case 3

Use case 3 allows the user to create a questionnaire, give it a layout, preview, answer and create a URL for the questionnaire. The questionnaire created in this use case has 5 pages: page 1 is a welcome page, then follow page 2, 3 and 4 with questions to which the user can answer and a 5th page which is the "Thank you for answering" page. Page 2 contains two questions with a single choice answer, page 3 contains a question with an answer matrix in which the chosen answers can be clicked and page 4 has a ranking question in which 10 points can be dispersed on to three choices. The content of the questions in the test specifications is made up. The difficulty in this test case was mainly to type Swedish characters with Sikuli. Typing Swedish characters, like "å" and "ä", is only possible with the "paste" function of Sikuli and a command "u" just before the string parameter signaling UTF-8 characters. For example: paste(u"å ä ö").

5.4 Use Case 4

Use case 4 of the AUT project is illustrated with the "phpMyAdmin" database and DecideIT. This use case assumes that DecideIT is linked to a database. The use case allows the user to create a select-statement with SQL code inside DecideIT, which then computes the results of the selection to weight alternatives in a decision tree. For example, the user who created the decision tree in figure 5.7 would like to decide between three test automation tools. The user would like to make an informed decision and decides thus to incorporate statistics about knowledgeable testers in the domain into his/her decision making. He/She consults a database and makes a first database selection for
the preference of testing tools amongst experts in the domain. Hence the user selects all testers using test automation in Sweden, with experience over three years and using an open-source testing tool. From this group of ‘experts’, a second selection filtrates and groups the three - by those experts - most used testing tools. An example SQL code that can be created could look like the code in figure 5.7. The user has another condition: it is important for the user that the tool can be used for web-, mobile-, and native program testing all together (“test_types”). On a scale from 0 to 100 value points, the user would therefore value tools which can serve all three kinds of test_types with 100 points, two kinds of test_types with 60 points and one type of test_types with 20 points.

The result of the selection enlightens the user that the three most used tools, judged by the selected expert group, are Robot Framework, Sikuli and another tool called "Other". 50% of the expert group use Sikuli as testing tool, 30% use Robot Framework and 20% use "Other". The database also delivers the information that RobotFramework can be used for all three types of testing, Sikuli just for web-, and native program testing and "Other" can just be used for one kind of testing. Let us assume DecideIT has a function, which gathers the results of the SQL selection, as well as the user value dispersion for the types of tests in a special syntax and imports the data automatically into a specified node of the DecideIT tree. Statistics could this way support the decisions of the user as exemplified in figure 5.7:

**Figure 5.7:** A mock-up display of incorporating SQL code in a decision tree
Additionally it was tested to run the scripts for use case 4 inside a for-loop in order to pass results from different SQL-requests into the same tree recursively. This is useful if the user would like to compare different calculations for one decision tree. For example the user decides that he/she wants to expand the 'experts group' to testers with 1 or more years of experience in a second calculation. In a third calculation the user wants to just have testers from Germany. The user starts with making a multidimensional array function with objects as shown in figure 5.8. The process looping through these objects, one at a time, starts in RIDE. The function 'sql_into_decideIT' returns a multidimensional array (an array of objects) in RIDE (line 1, figure 5.9). RIDE and Sikuli can both read multidimensional arrays, but they cannot pass multidimensional arrays to each other, only Strings. During the first for-loop iteration, RIDE loops over the first row in the multidimensional array and transforms it into a string called 'oneRowString'. This transformation is done with help of a data format called 'json' and its command 'Dumps' (line 3). This 'oneRowString' is read by Sikuli and converted back into an object with three values ('is_open_source': true, 'country': 'Sweden', 'experience': 1). Sikuli inserts the three values into the three WHERE-comparison statements in the SQL-command. The SQL-command then generates the first decision tree in DecideIT. During the second for-loop iteration, the next row is read by RIDE, passed to Sikuli and Sikuli inserts the corresponding values into the SQL command, just as before. Consequently the second decision tree with new calculations is generated and displayed next to the first tree.

The for-loop implementation is an own test case (use case 3, test case 3.0, appendix E) and constitutes an alternative flow (the for-loop is for voluntary use) which is why it is marked green in the test specifications.

---

Figure 5.8: A multi-dimensional array function

Figure 5.9: A for-loop example in RIDE which passes a multidimensional array to SIKULI

---

13 the variable $ {row} in line 2 is an object in the multidimensional array
5.5 Use Case 5 and Use Case 6

Use case 5 allows the user to create an email message from an existing template. The user can then customize the layout for the email. In this use case, the user gives the email a header, a body and inserts an image. The user can then test send the email to a Google account and verify if the email is displayed correctly in the web-browser. Use case 6 allows the user to view, save and archive a report for calculations made on a decision tree. Use Case 5 and 6 are particularly noteworthy, since they show that RFSI are able to successfully mix two types of tests: testing a native program and testing inside a web-browser.

5.6 CleanUp

After creating and stabilizing all use cases, "CleanUp" scripts were created with the intention to be run after each use case. The goal of the "CleanUp" scripts is to clean the remainders of failed test runs, such as open windows or saved test files which have to be deleted before the next test run. First one clean up script was constructed to rinse rests of all kinds of test case fails, but was then split up into several different clean up scripts. This was done to save execution time, since the "CleanUp" for a report script cleanses reminders which are unique for the report script and it is therefore enough to run that "cleanse" after a report use case has finished running, instead of running those lines after each use case.

5.7 Observations during implementation

During implementation of the test scripts, it was also discovered that Sikuli test scripts can only be run from the beginning, not from the middle of a script. This raises waiting times while debugging and therefore increases maintainability. This observation was deemed important and was therefore added to the discussion, although not part of any theory or metric.

Unfortunately it was also observed that the RIDE logs were not displayed anymore after a number of test runs and on other few occasions it was the case that RIDE froze and would not respond to any actions anymore. Those incidents were counted as crashes and since the number of RIDE crashes was measureable, they were subsequently added to the metrics and the results, although not found as a criteria in literature, since it is implied that a functional testing tools preferably should never crash.
Chapter 6

Results

In this chapter it is evaluated to which degree the presented requirements for successful automated regression tests could be implemented via remote desktop for tests with RFSI (Robot Framework with Sikuli integration). The use cases and test steps are described in appendix D. A general summary of the metrics and results for the evaluation can be found in appendix B. A summary of all test statistics can be seen in appendix F.

Although the research and tests are written for the AUT project, all use cases, test steps, screenshots, illustrations and explanations for the practical implementation of tests that are covered in the thesis will convey examples from applications other than AUT for non-disclosure reasons. The use cases, test steps, examples and explanations chosen for illustration purposes are selected to match the functionality covered in the AUT project and the working principles behind writing the tests in RFSI are the same for the AUT project implementation of scripts as for the examples given in the thesis.

6.1 Functionality requirements

RFSI based tests are able to mimic the actions of a user without restriction. An example of functionalities tested is "click", "drag-and-drop" and "type".

Sikuli has approximately 120 functions for testing and Robot Framework was used with two libraries containing about 150 functions. Of those together 270 functions, only 36 were needed to implement all test steps in the test specifications.

A first metric for functionality in the ISO/IEC is "functional adequacy", which measures the number of tool-inherent functions that are suitable for performing the specified test steps. Functional adequacy is measured by taking the ratio of the number of functions which showed limitations within the project context (inadequate functions) to the total number of functions implemented. This ratio is subtracted from 1 to get the percentage of adequate functions (functional adequacy). The functional adequacy of RFSI is 97%, which means of all 36 functions used to implement test steps, only one problem was encountered with the type() function when typing Swedish characters (see chapter 5.3).

The functional implementation of RFSI measures the number of limitations encountered during the implementation of self-written functions in relation to the number of implemented functions, for the purpose of RFSI customization. Limitations refers to functionality requirements which could not be implemented because of constraints in RFSI integration. This measurement gives an idea about the limitations of customizing RFSI for the testers purposes. No limitation has been encountered while implementing 11 customized functions. The functional implementation was therefore 100%.

The functional implementation coverage is a full 100%, i.e. all 637 test steps in the
test specifications were implemented.

RIDE offers "play", "stop", "pause", "continue" and "step over" buttons, which facilitate the handling of test executions (see appendix G). It is possible to log, trace back and report details of test failure with the Robot Framework log and report functionality. Separation of test data from test scripts is a built-in functionality in Robot Framework and diverse keywords are designed for this concept, such as keywords for cleaning test data or changing test data format.

6.2 Reliability requirements

The reliability for false positives shows how many times the testing tool shows a positive test run, when the actual test result was negative. It is measured by taking the ratio between the number of false positives relative to the total number of test runs (percentage of unreliability) and subtracting the result from 1 to get the percentage of reliability. A reliability of 99.7% was calculated for false positives. One false positive occurred during all 404 testing runs and was due to the testers accidental intervention.

The reliability for false negatives shows how many times the testing tool shows a negative test run, when the actual test result was positive. The reliability for false negatives after stabilization is 95.4%, which corresponds to 1 out of 20 test runs.

Robot Framework and Sikuli have the capability to handle their own system errors and failures. They automatically restore their system, log the error/failure and no data was ever destroyed or damaged.

6.3 Maintainability and stability requirements

Activity recording serves the purpose of facilitating the understanding of a flow of activities of a test script by logging those activities to the test log. This measure is done by counting the number of test steps logged in the test log and comparing it to the total number of test steps in the test specifications. The recording of test steps in the test log is 59%, which means that more than every second test step (of a total of 637) is recorded in the test logs. This test activity is considered high, since the description of most test steps is not necessary for the understanding of an activity flow; for example, many test steps describe waiting activities (the tool waits for the AUT to react). 59% recording of activities results in high maintainability, since test steps can be followed thoroughly.

The MCC (McCabe cyclomatic complexity) describes how complex the test scripts are concerning the use of decision elements, i.e. the occurrence of logical AND and OR operators in the test scripts (for example if-statements and for-loops). 101 decision elements occurred in total and the average MCC of all test is 5.2, while the highest MCC observed in a test script was 10 and the lowest was 0. An MCC under 10 is recommended for all test scripts (see chapter 3.8.3). The test scripts are thus not complex concerning decision elements. Maintainability is therefore deemed high, since the structure of the test scripts is relatively easy to understand.

A change of background color in (elements of) the AUT, a pop-up window covering sight of (elements of) the AUT and a change in AUT window size can be cause for failure of test script execution. The test scripts are not stable concerning the potential
AUT modifications mentioned above. A change in element location (as long as in sight) does not disturb the tests at all.

The defect trend measure shows the percentage of test scripts fails over 7 days of test writing and stabilization. The defect trend measure (figure 6.1) illustrates that the percentage of failed tests declines the longer test script development time goes. This is expected and desirable; it shows that the number of false negatives shrinks as tests become more and more stable. All six use cases could be stabilized to 95% of successful test runs after at most seven test and stabilization days per use case. The tests can be considered 95% stable.

![Defect Trend Measure](image)

**Figure 6.1:** The defect trend measure shows that the percentage of failed tests diminishes over the number of test days

In total 5 RIDE crashes occurred in a number of 404 test runs, two of them were RIDE log crashes (the logs were not displayed anymore) and three crashes froze the RIDE interface. No Sikuli crash occurred. The reliability concerning crashes is 99% for Robot Framework and 100% for Sikuli. Anything below 100% is considered critical, since
crashes in what are considered functional testing tools could result in loss or corruption of important data.

### 6.4 Efficiency requirements

The measurements for efficiency beneath were taken with an inexperienced tester with an unknown tool for a newly encountered AUT project. Those measurements are relative and depend on individual knowledge and experience. The purpose of those metrics was to serve as a point of reference for future work in this domain or for project managers who would like to estimate testing efforts for similar projects.

The average number of lines of code written per day was 106 and corresponded to one use case in average. 54 test runs per use case were needed until the tests could be considered 95% stable, which corresponds to 2-3 working days (8h) of only running and stabilizing tests. With those statistics, the maximum number of test suite runs possible per day was 10 test suite executions. The test execution time of the whole test suite is around 48 minutes whereof the testing tool has to wait about 15 minutes for AUT execution-, loading- and deletion operations. The total number of lines of code in the test suite including the "CleanUp" scripts amounts to approximately 1949 lines of code, which are executed in about 33 minutes (taking away AUT operation time). This means test execution time for one line of code is around 1.02 seconds. Further test statistics can be seen in appendix F.

It was estimated by a system expert that 85% of all test needs for the AUT had been implemented, which took 21 working days and corresponds to 168 hours. The extrapolated time to implement tests for the whole AUT was calculated to amount to 197.65 hours. The system development time of the AUT by programmers was reported to be close to 500 hours. The total test time in relation to the total development time for the AUT is thus 39.53%. ISTQB recommends that an additional 30-40% of development time is spent on testing, which was the case for the proof of concept (see chapter 3.8.4).

### 6.5 Understandability requirements

Naming conventions were followed for all test implementations and logs. For example small letters were used for functions, keywords are separated by an underline in Python scripts, test scripts were given the same names as test cases in capital letters and test activity traces were given the same number as for test steps in the test specifications.

The error location measures the number of execution traces (comments in the scripts). This measure is similar to the activity recording measure with the difference that the number of comments in scripts which also can be retrieved in a test log are counted in activity recording. An error location in a script can be found easily, since 62% of all test steps are traced. There are more comments within a test script than comments passed to a test log, since it is vital to not only comment test step activities, but also the meaning

---

141 working day = 8 hours  
15underlines are translated to empty spaces in RIDE when those keywords are used  
16with two expections for the questionnaire since the test script would have been too long otherwise
of functions and decision elements in order to facilitate debugging.

Verification has been implemented at the end of each test script by writing "OK" to an external file after each successful test script execution. If a Sikuli test script would fail, the "OK" assertion would not be written to the file and Robot Framework consequentially marks that test script as "Fail: AssertionError" in the log.

Documentation, tutorials and examples can be found in abundance for Robot Framework and Sikuli. For Robot Framework and Sikuli integration, this pilot study can be considered the first analysis with measurements.

6.6 Portability requirements

Both testing tools are platform and OS independent, since they execute Python and java based scripts\(^\text{17}\).

API’s, plug-ins and additional modules exist for both tools, for instance the Java API for Robot Framework.

Robot Framework, which runs the test scripts, is compatible with building tools such as Maven, and can thus be integrated into an automated chain of building, deployment and regression testing, triggered periodically or by code updates.

\(^{17}\)Python is a platform and OS independent language
Chapter 7

Discussion

In this chapter, the results of the thesis are scrutinized, the questions of the thesis are answered and RFSI recommendations are given for different contexts. Finally future work and possible ethical and social implications of this study are discussed.

7.1 Metric selection

Many more metrics than the ones chosen were available in literature about software quality measurements and many would probably have served the purpose of tool and test script evaluation just as well. However the metrics had to be selected and adapted to fit the scope and the context of the proof of concept. This selection was done according to the criteria presented in the methodology chapter. The metric for functional implementation might have been more representative for judging the limits of RFSI if more than 11 functions were implemented for RFSI customization, but it might serve as a point of reference.

7.2 Degree of requirement fulfillment

Reliability was only partially fulfilled due to RIDE crashes. Although just one out of 81 test runs crashed, crashes should ideally not occur at all. Nevertheless, RIDE had the capability to restore all of its features and no data was lost or corrupted.

The requirement for maintainability and stability were partially satisfied. The fact that test runs can fail because of changes in the GUI of the AUT, such as background color or window size, is likely to be a disadvantage when considering the use of RFSI as tool in test projects. A project manager might want to take this into account. The visual nature of the RFSI tests imply that the AUT has to be in the foreground during test runs, which means that either a dedicated computer has to be used or the tests have to be run over night.

Testers also need to take into consideration that Sikuli test scripts can only be run from the beginning, not from the middle of a script. Therefore, maintenance and debugging would be facilitated by keeping test scripts small.

Efficiency was partially fulfilled. The test execution time can be considered very slow compared to executing conventional application code. However, the estimation of execution time of 1.02 seconds is rough at best. It is based on total test suite execution time per total lines of code. But included in the execution time are several wait() functions, which were necessary to give the AUT time to respond. This significantly increases the
total test execution time\textsuperscript{18}. Actual test execution time is therefore estimated to be much faster. At any rate, automated test execution with RFSI over the remote desktop is faster than manual testing. It also has the advantage of being logically structured, repeatable and automatically delivering test reports, logs and statistics.

Unfortunately no similar research was found to be compared with this proof of concept and therefore no comparison to other results was possible. From earlier experience with the open-source tools Selenium and MonkeyTalk, the results are judged to be plausible; for example an execution time per line of code close to one second was also observed for MonkeyTalk in tests of a mobile application.

7.3 Thesis answers

The thesis contributes as pilot study to the domain of automated software testing by evaluating if RFSI (Robot Framework with Sikuli integration) is a viable alternative to proprietary testing tools for automated GUI regression tests over a remote desktop. The first goal of this proof of concept study was to define internal and external requirements for automated GUI tests over the remote desktop, which was done through a literature study and presented in chapter 3.8. A second goal of this thesis was to evaluate to which degree the tool RFSI and RFSI-based test scripts fulfill the presented requirements in tests of GUI desktop applications over a remote desktop. The results of the proof of concept study showed that the internal and external requirements for successful automated testing were completely fulfilled for functionality, understandability and portability. The requirements were partially fulfilled for reliability, maintainability and stability, and efficiency. A third goal of this research was to determine if RFSI can be considered an alternative to proprietary tools for testing GUI desktop applications over a remote desktop. In conclusion, the results indicate that RFSI fulfills enough requirements to be considered a viable alternative to proprietary testing tools for similar projects.

7.4 Test tool recommendation

RFSI is not recommended for tests of large scale applications and life-critical systems. According to the test execution times measured, test time might exceed 8 hours for test suites over 32 000 lines of code, which is prone to become a problem for tests of those kind of large scale applications. Another drawback is the reliability for false negatives of 95\%, which is not enough for testing life-saving systems.

However, RFSI is recommended for non-safety critical middle sized to small software applications\textsuperscript{19} with a relatively long lifecycle and projects which need rapid feedback of the main functionalities of an AUT. It possesses functionality sufficient for test needs\textsuperscript{20}, it is easily understood, able to mix web - and desktop application testing, the tests scripts are quickly written, maintained and relatively stable within a short time.

\textsuperscript{18}not because of the execution capability of RFSI, but because of the slow execution time of the AUT

\textsuperscript{19}under 30 000 lines of code

\textsuperscript{20}as far as the project study determined


7.5 Ethical and social aspects

If this research would lead to the introduction of RFSI in testing projects in the future, it has to be kept in mind that most testers still work manually. Many of them have years of experience with manual testing and have acquired a status in this domain. The introduction of a new technology might threaten their position, and, in the worst case, fear of job-loss might arise. This is especially the case, since the use of RFSI potentially facilitates offshoring and outsourcing of testing resources: RFSI is likely to reduce testing costs and does not open for attack surfaces as proprietary tools do with the introduction of agents when testing via remote desktop. This technically simplifies the concept of "Testing as a Service" (TaaS), which is often offered by big offshore companies. As mentioned in the introduction, India is already one of the leading destinations for companies fostering this business opportunity. Nevertheless, it is a possibility that the use of open-source tools for testing could also lead to more companies encouraging automated testing from within, because of the ease of use of open-source tools and their relatively fast results. At any rate, the use of open source tools for testing is believed to further reduce manual testing in the future. Communication and openness about the consequences of introducing RFSI into the testing process are recommended.

7.6 Future Work

For future research in this area, a direct comparison of RFSI to commercial and other testing tools, including a Return on Investment (ROI) analysis would give strategical information if and how to use RFSI on a larger scale. Furthermore, a usability analysis would give hints on how fast RFSI can be learned and applied by its users.

It is likely that the open source community will produce further innovative tools with improved capabilities and ease of use in the future - automated testing is therefore probable to become a more and more common practice and worth observing.
Bibliography


BIBLIOGRAPHY


Appendices
Appendix A

Robot Framework and Sikuli IDE

Figure A.1: The Robot Framework IDE

Figure A.2: The Sikuli IDE
Appendix B

Requirement specifications metrics for successful automated testing

The tables are constructed according to the layout for metrics in ISO/IEC-TR9126-3.
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional adequacy</td>
<td>How adequate are the used functions for implementing the test steps?</td>
<td>Count the number of tool-inherent functions that are not suitable for performing the specified test steps, then measures the ratio of it to functions implemented without problems.</td>
<td>$X = 1 - \frac{A}{B}$&lt;br&gt;A= Number of functions in which problems are detected in evaluation.&lt;br&gt;B= Number of functions implemented; B= 36;&lt;br&gt;0 ≤ X ≤ 1; the closer to 1, the more adequate</td>
<td>Sikuli Scripts; Test specifications</td>
</tr>
<tr>
<td>Functional implementation</td>
<td>How many limitations are encountered during the functional implementation?</td>
<td>Count the limits encountered while implementing the self-written function in relation to the number of self-written functions.</td>
<td>$X = 1 - \frac{A}{B}$&lt;br&gt;A= Number of limits encountered while implementing functions;&lt;br&gt;B= Number of self-written functions; B= 11;&lt;br&gt;0 ≤ X ≤ 1; the closer to 1, the more complete</td>
<td>common Functions Script Script code; Requirements (chapter 3.8.1);</td>
</tr>
</tbody>
</table>
### Metrics for Functionality

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional implementation coverage</strong></td>
<td>How correct is the functional implementation?</td>
<td>Count the number of test steps that could not be implemented and compare with the number of test steps described in the test specifications</td>
<td>$X = 1 - \frac{A}{B}$</td>
<td>Sikuli Scripts; Test specifications</td>
</tr>
<tr>
<td></td>
<td>A= Number of NOT implemented test steps. n</td>
<td>A = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B= Number of test steps described in test specifications</td>
<td>B= 637</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 ≤ X ≤ 1; the closer to 1, the more covered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RIDE menu</strong></td>
<td>Is it possible to start, pause and stop the test execution?</td>
<td>Check if functionality exists in Robot Framework</td>
<td>-</td>
<td>RIDE</td>
</tr>
<tr>
<td><strong>Check test failure</strong></td>
<td>Is it possible to log, trace back and report details of test failure?</td>
<td>Check if functionality exists in Robot Framework</td>
<td>-</td>
<td>RIDE</td>
</tr>
<tr>
<td><strong>Test data separation</strong></td>
<td>Is it possible to separate test data and scripts?</td>
<td>Check if functionality exists in Robot Framework</td>
<td>-</td>
<td>RIDE</td>
</tr>
</tbody>
</table>
### Table B.2: Requirement metrics for Reliability

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>False positives</strong></td>
<td>How many times does the tool show a positive test run, when the actual test result was negative?</td>
<td>Count the number of times when a false positive was shown after a test run, relative to the total number of test runs</td>
<td>X = 1 - ( \frac{A}{B} )</td>
<td>( X = 1 - \frac{1}{404} ) = 0.997 = 99.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A= Number of test runs with false positives</td>
<td>A = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B= Total number of test runs</td>
<td>B = 404</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 ≤ X ≤ 1; The closer to 1, the more reliable not to show a false positive</td>
<td></td>
</tr>
<tr>
<td><strong>False negatives</strong></td>
<td>How many times does the tool show a negative test run, when the actual test result was positive? Note! Not included: test script mistakes before stabilization!</td>
<td>Count the number of times when a false negative was shown after a test run, relative to the total number of test runs</td>
<td>X = 1 - ( \frac{A}{B} )</td>
<td>( X = 1 - \frac{6}{131} ) = 0.954 = 95.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A= Number of test runs with false negatives</td>
<td>A = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B= Total number of test runs after stabilization</td>
<td>B = 131</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 ≤ X ≤ 1; The closer to 1, the more reliable not to show a false negative</td>
<td></td>
</tr>
<tr>
<td><strong>Error/Failure handling</strong></td>
<td>Are RIDE and Sikuli able to handle their own system errors and failures?</td>
<td>Check if functionality exists</td>
<td>-</td>
<td>End state of RIDE and Sikuli</td>
</tr>
</tbody>
</table>
### Metrics for Maintainability and Stability

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
</table>
| **Activity recording** | How thorough is the recording of test activities in the test log? | Count the number of test steps logged in the test log and compare it to the total number of test steps in the test specifications. | Sikuli Scripts; Test specifications | \( X = \frac{A}{B} \)  
\( A = \text{Number of implemented test step logs} \)  
\( A = 377 \)  
\( B = \text{Total number of test steps} \)  
\( B = 637 \)  
\( 0 \leq X \leq 1; \)  
the closer to 1, the more activities can be traced;  
\( X = 1 - \frac{377}{637} = 0.59 = 59\% \)  
More than half of all test steps are logged |
| **MCC** | How many independent basic paths exists in one test script? | Count the number of decision elements, i.e. a condition with some occurrence of the logical operators AND and OR; | Value \( \text{DE taken from scripts} \)  
\( \nu(G) = \text{DE} + 1 \)  
\( \text{DE} = 4.2 \)  
MCC should be \( \leq 10 \) for all test scripts  
The smaller the MCC, the less complex the scripts  
\( \nu(G) = 4.2 + 1 = 5.2 \leq 10 \)  
which is desirable |
# Metrics for Maintainability and Stability

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>How stable are the test scripts when exposed to chosen incidents I?</td>
<td>Check if scripts go through when incident occurs</td>
<td>I1 = Change background color I2 = pop-up window which covers sight on AUT I3 = Change location of buttons I4 = Change AUT window size</td>
<td>Sikuli Scripts RIDE AUT</td>
<td>I1 = No I2 = No I3 = Yes I4 = No</td>
</tr>
<tr>
<td>What is the percentage of failed test runs after a period of stabilization?</td>
<td>Count percentage of test scripts which fail a test run over nr of days</td>
<td>Goal: Only ≤ 5% of scripts may fail for tests to be considered stable</td>
<td>Measurements see table in chapter 6.3</td>
<td>All scripts were ≥ 95% stable after a maximum of 7 days per use case</td>
</tr>
<tr>
<td>How many times did RIDE and Sikuli crash during the development time?</td>
<td>Count the number of times the test tool crashed during the development time, relative to the total number of test runs</td>
<td>( X = 1 - \frac{A}{B} ) ( Y = 1 - \frac{C}{B} ) ( A = ) Number of RIDE crashes ( A = 5 ) ( B = ) Total number of test runs ( B = 404 ) ( C = ) Number of Sikuli crashes ( C = 0 ) ( 0 \leq X \leq 1; ) The closer to 1, the more reliable not to crash</td>
<td>End state of RIDE and Sikuli</td>
<td>( X = 1 - \frac{5}{404} = 0.99 = 99% ) ( Y = 1 - \frac{0}{404} = 1= 100% )</td>
</tr>
</tbody>
</table>
### Table B.4: Requirement metrics for Efficiency

<table>
<thead>
<tr>
<th>Metric for Efficiency</th>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Writing Efficiency</td>
<td>What was the average nr of test steps implemented per day?</td>
<td>Count average nr of lines written over the total of days in which code was written</td>
<td>$\frac{\text{Lines of Code}}{\text{Nrof Days}}$</td>
<td>Test scripts; Measurements</td>
<td>106 $\frac{\text{Lines of Code}}{\text{Day}}$ corresponds to 1 $\frac{\text{Use Case}}{\text{Day}}$</td>
</tr>
<tr>
<td>Test Runs for Stabilization</td>
<td>What was the average nr of test runs needed to stabilize a use case?</td>
<td>Count test runs until use case was considered stable ($\leq 5%$ of use cases fail)</td>
<td>$X = \frac{\text{Nrof test runs}}{\text{Use case}}$</td>
<td>Test Logs; Measurements</td>
<td>54 $\frac{\text{Test Runs}}{\text{Use case}}$ or $\approx 2 - 3$ working days ($8h$)</td>
</tr>
<tr>
<td>Max nr of test runs per day ($8h$)</td>
<td>What is the maximum of test suite runs possible per day?</td>
<td>Count average execution time for test suite execution</td>
<td>$\frac{\text{Hours per Day}}{\text{Test Suite#x: Time (h)}} = \text{nr of test suites executed}$</td>
<td>Test Logs; Measurements</td>
<td>$480 \text{ min} = 10$ $\frac{\text{Test Suite#}}{\text{Day}}$ executions are possible during $8h$ (1 working day)</td>
</tr>
</tbody>
</table>
## Metrics for Efficiency

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>How long would it take to automate all test requirements in the AUT relative to development time for new functions in the AUT?</td>
<td>Count percentage of AUT automized and extrapolate; Get info about development time</td>
<td>$A = \frac{\text{Percent of AUT test requirements automated during 21 working days (168 h)}}{\text{Time for Test Suite Execution (h)}}$</td>
<td>Expert judgement; Measurements</td>
<td>The time for automation is 39.53%, which is within 30-40% of development time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A = 85%$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$B = \text{Hours for development of new functions in AUT}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$B = 500$ h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$30-40%$ of $B = 150-200$ h;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$C = \text{Automation of 100% of AUT}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$C \approx 25$ days or $197.65$ h</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Testing time should be between 30-40% of development time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table B.5: Requirement metrics for Understandability

<table>
<thead>
<tr>
<th>Name conventions</th>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are naming conventions being used?</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Test scripts;</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Understand error location**

- **How many execution traces are used per test steps?**
  - **Count execution traces**
  - **Formula:** \( X = \frac{\text{NrTraces}}{\text{NrTestSteps}} \)
    - **Results:** \( X = \frac{398}{637} = 0.62 = 62\% \)

<table>
<thead>
<tr>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is verification being used?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation, tutorials, examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does documentation, tutorials and examples exist for both Robot Framework and Sikuli?</td>
</tr>
</tbody>
</table>
### Table B.6: Requirement metrics for Portability

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Application</th>
<th>Formula</th>
<th>Source of metrics input</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform and OS independence</td>
<td>Are the testing tools platform and OS independent?</td>
<td>-</td>
<td>Test scripts; Yes, both are Python based and therefore platform and OS independent</td>
<td></td>
</tr>
<tr>
<td>API, Plugin module</td>
<td>Do additional API’s, add-ins, plug-ins or modules exist for both tools?</td>
<td>-</td>
<td>Test scripts; Yes, f.ex: RF Java API, Jenkins and Maven plug-in</td>
<td></td>
</tr>
<tr>
<td>Compatibility with building tools</td>
<td>Is Robot Framework compatible with a building tool?</td>
<td>-</td>
<td>Test scripts; Yes, with Maven for example</td>
<td></td>
</tr>
</tbody>
</table>
Figure C.1: The GQM model for the proof of concept
Appendix D

Use cases and test specifications

D.1 Use case 1: Create, built, execute and delete a DecideIT decision tree

Brief Description

Use case 1 allows the user, a decision analyst, to create a basic decision tree in the program DecideIT. Decision trees in DecideIT6 can serve as a basis for the decision analyst to evaluate the decisions and risks associated with different events and possibilities.

Actors

Primary actor: decision analyst; Robot Framework
Secondary actor: DecideIT

Basic Flow

1.1 CREATE NEW DECISION TREE
This use case starts a decision tree framework. The user clicks on “New” and chooses the option “Decision Tree”, validates the number of nodes and confirms

1.2 BUILD UP DECISION TREE
This use case starts with a framework of a decision tree in the view field. The user names the tree and the two main options. For the first main option, the user creates an event with three nodes. Afterwards the user gives the nodes names, probabilities and values within two intervals.

1.3 EXECUTE DECIDE IT
This use case starts when the example decision tree is built up with two main nodes and an event on one of the main nodes. The event has three nodes which have been defined in 1.2. The user chooses to calculate a cumulative profile, a risk profile and a value tornado for the decision tree.

1.4 CLEAN UP OF DECIDE IT
The user executes an example deletion of a node and then closes the decision tree window without saving the created decision tree in order to start the same use case again.
## D.2 Test specification for use case 1

<table>
<thead>
<tr>
<th>Use case # / Preconditions</th>
<th>Test case and Test steps</th>
<th>Test case/step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USE CASE 1:</strong> Create, build and execute a dialog</td>
<td>1.0</td>
<td>Create new DeciDelT file</td>
<td>A decision tree framework is built up</td>
</tr>
<tr>
<td>In desktop view</td>
<td>1.1</td>
<td>Click on: File→ New</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>Wait for pop-up dialog to appear</td>
<td></td>
</tr>
<tr>
<td>In pop-up dialog window</td>
<td>1.3</td>
<td>Choose Option: Decision Tree</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>Click on: OK</td>
<td></td>
</tr>
<tr>
<td>DecisionTree has to be in view</td>
<td>2.0</td>
<td><strong>Build example decision tree</strong></td>
<td></td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.1</td>
<td>Click on bar “Operation types”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In Node properties window</td>
<td>2.3</td>
<td>Click on first name field: Type Tree name: UseTestAutomation?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>Click on: OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>Wait until Text has shown up in Decision tree</td>
<td></td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.6</td>
<td>Click on: Field A1</td>
<td></td>
</tr>
<tr>
<td>In Node properties window</td>
<td>2.7</td>
<td>Click on: Field: Alternative 1</td>
<td></td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.8</td>
<td>Type: yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>Click on: OK</td>
<td></td>
</tr>
<tr>
<td>In Node properties window</td>
<td>2.10</td>
<td>Click on: Field A2</td>
<td></td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.11</td>
<td>Click on: Field: Alternative 2</td>
<td></td>
</tr>
<tr>
<td>In Node properties window</td>
<td>2.12</td>
<td>Type: no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.13</td>
<td>Click: OK</td>
<td>All three fields are named</td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.14</td>
<td>Click on: C1</td>
<td></td>
</tr>
<tr>
<td>In Node properties window</td>
<td>2.15</td>
<td>Wait for pop-up dialog to appear</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.16</td>
<td>Click on: Nr of sub-nodes</td>
<td></td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.17</td>
<td>Type: 3</td>
<td>An event with three nodes is built on what was node A1</td>
</tr>
<tr>
<td>In Node properties window</td>
<td>2.18</td>
<td>Click: OK</td>
<td></td>
</tr>
<tr>
<td>In Tree view</td>
<td>2.19</td>
<td>Wait for Tree view to appear</td>
<td></td>
</tr>
<tr>
<td>In Probability view</td>
<td>2.20</td>
<td>Click on: C1 Dialog field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.21</td>
<td>Click on: Outcome field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.22</td>
<td>Type: Robot Framework</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.23</td>
<td>Click on: Probability tab</td>
<td></td>
</tr>
</tbody>
</table>
2.24 Click on: Most likely outcome field
2.25 Type: 40
2.26 Click on: Value view

**In Value view**
2.27 Click on: second circle-> setting interval without most likely point

2.28 Click on: Interval Min. field
2.29 Type: 20
2.30 Click on: Interval Max. field
2.31 Type: 40
2.32 Click: OK
2.33 Wait for Tree view to appear

**In Tree view**
2.34 Click on C2 Dialog field
2.35 Click on: Outcome field
2.36 Type: Sikuli
2.37 Click on: Probability tab
2.38 Click on: Most likely outcome field
2.39 Type: 50
2.40 Click on: Value view

**In Value view**
2.41 Click on: second circle-> setting interval without most likely point

2.42 Click on: Interval Min. field
2.43 Type: 40
2.44 Click on: Interval Max. field
2.45 Type: 60
2.46 Wait for Tree view to appear

**In Tree view**
2.47 Click on C3 Dialog field
2.48 Click on: Outcome field
2.49 Type: Own Code
2.50 Click on: OK
2.51 Click on: Value view
2.52 Click on: first circle-> most likely point
2.53 Type: 45
2.54 Click on: OK

**In Tree view**
3.0 **Execute DecideIT**

**Different profiles are calculated**

3.1 Click on: Cumulative Profile icon

3.2 Wait for pop-up window with cumulative profile to appear
3.3 Click on "X" icon to close the window
3.4 Click on: Risk Profile icon
3.5 Wait until Risk Profile window has appeared
3.6 Click on "x" icon to close the window
3.7 Click on: Value Tornado icon
3.8 Wait until Value Tornado window has appeared
3.9 Click on "x" icon to close the window

4.0 Run Clean up DecideIT

| The newly built DecideIT tree has been deleted |

4.1 right click on the event node C3

4.2 choose "delete node"

4.3 Click on: OK in the pop-up window which asks if the user is sure to want to delete

4.4 Click on: OK in error message window which says that the probabilities have to be rewritten

4.5 Rewrite probabilities in new pop-up window

4.8 Click on: OK

4.7 Close DecideIT
D.3 Use case 2: Build, add and execute additional decision trees

Brief Description

Use case 2 and the corresponding test specifications in this appendix have been made up in order to match the test steps for the AUT project. To execute the test steps as described in the specification for use case 2 is therefore not possible in the program DecideIT\(^1\), but for examplification of the functionalities tested in use case 2 of the AUT, let us assume the test steps described here can be done. Let the decision tree from use case 1 be called the "main tree". Use case 2 allows the user, a decision analyst, to build and add an additional decision tree to one of the branches of the main tree created in use case 1 in the program DecideIT. Use case 2 describes the test steps which are necessary in order to recursively build and add a selection of those additional trees to the main tree and then execute calculations on the additional tree. In total, it was requested to test building and adding 5 of 23 possible different additional trees to the main tree and execute calculations on those additional trees. To build, add and execute one additional tree comprises in average 34 test steps.

Actors
Primary actor: decision analyst; Robot Framework
Secondary actor: DecideIT

Basic Flow

2.1 BUILD AND ADD ADDITIONAL DECISION TREE
This use case selects the 'add additional tree' button in the menu and clicks on one of the branches of the main tree to which the additional tree shall be added. The user then customizes the choices in the pop-up window for the additional tree, such as for printing reports, number of nodes, etc. This use case starts with a framework of a decision tree in the view field. The user names the tree and its main options. The user may create new events with several nodes. Afterwards the user gives the nodes names, probabilities and values within two intervals. Before execution the tree has to be saved in a certain folder, to which the user navigates.

2.2 EXECUTE ADDITIONAL AND MAIN TREE
This use case starts when the main and additional decision tree is built up and all probabilities, values etc. are set. The user chooses to calculate a cumulative profile, a risk profile and a value tornado for the decision tree and to print out reports for all of those profiles.

\(^{21}\)it is though possible to augment an existing tree with additional branches
D.4 Test specification for use case 2

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Test case # and Test steps</th>
<th>Test case/step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a main tree</td>
<td>1.0</td>
<td>Build and add an additional Decision tree to the main tree</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Find Main Tree Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>Push button &quot;Add additional tree&quot; in the menu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Click on Main Tree operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In 'select additional tree' window</td>
<td>1.4</td>
<td>Drag line from 'Main Tree' to empty space</td>
<td>Select additional tree window shows up</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>Select 'tree with 2 nodes'</td>
<td></td>
</tr>
<tr>
<td>In Group properties window</td>
<td>1.7</td>
<td>Click on OK</td>
<td>Group properties window shows up</td>
</tr>
<tr>
<td></td>
<td>1.8</td>
<td>Type: Additional Tree &quot;Tree name&quot; field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.9</td>
<td>Click on OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.10</td>
<td>If tree can be added to the selected main tree branch -&gt; 1.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.10.1</td>
<td>If tree cannot be added to the selected main tree branch -&gt; choose new branch or modify main tree branches so that additional tree can be added</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.11</td>
<td>Click on first tree branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.12</td>
<td>Name first tree branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.13</td>
<td>Give probabilities and values within a range</td>
<td></td>
</tr>
<tr>
<td>In pop-up window</td>
<td>1.14</td>
<td>Click on OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.15</td>
<td>Name second tree branch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.16</td>
<td>Give probabilities and values within a range</td>
<td></td>
</tr>
<tr>
<td>In Group properties window</td>
<td>1.17</td>
<td>Click on OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.18</td>
<td>Click on: Save tree</td>
<td></td>
</tr>
<tr>
<td>1.19</td>
<td>Navigate through folders to find the folder 'MainAndAddTree' in which the tree should be saved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.20</td>
<td>If folder does not exist -&gt; make folder 'MainAndAddTree'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Action</td>
<td>Result</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------------</td>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td><strong>Execute Calculations with additional tree</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Click on additional decision tree image (unique for each additional decision tree)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Click on: Cumulative Profile icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Wait for pop-up window with cumulative profile to appear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Click on 'Print out report'</td>
<td>Report appears in New Tab</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>Click on &quot;x&quot; icon to close the window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>Click on: Risk Profile icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Wait until Risk Profile window has appeared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>Click on 'Print out report'</td>
<td>Report appears in New Tab</td>
<td></td>
</tr>
<tr>
<td>2.9</td>
<td>Click on &quot;x&quot; icon to close the window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.10</td>
<td>Click on: Value Tornado icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.11</td>
<td>Wait until Value Tornado window has appeared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.12</td>
<td>Click on 'Print out report'</td>
<td>Report appears in New Tab</td>
<td></td>
</tr>
<tr>
<td>2.13</td>
<td>Click on &quot;x&quot; icon to close the window</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**4 other** Additional Trees
D.5 Use case 3: Create a questionnaire, layout, preview and answer it

Brief Description

The functionality of use case 3 and the corresponding test specifications in this example are a close match to the actual test steps for the AUT project. This use case allows the user, a business analyst, to create a questionnaire, give it a layout, preview and answer the questionnaire. The questionnaire has 5 pages: page 1 is a welcome page, then follow page 2, 3 and 4 with questions to which the user can answer and a 6th page which is the "thank you" page. Page 2 contains 2 questions with a single choice answer, page 3 contains a question with an answer matrix in which the chosen answers can be clicked, page 4 has a ranking question in which 10 points can be dispersed on to three choices. The content of the questions in the test specifications is made up.

Actors
Primary actor: decision analyst; Robot Framework
Secondary actor: Questionnaire template

Basic Flow

3.1 CREATE QUESTIONNAIRE WITH 3 QUESTIONS
The user chooses "New dialog", specifies the name and choses from enabling options.

3.2 FIRST QUESTION For the first question page, the user creates a first question "Do you have a mobile?". This question has two options "yes" and "no". The user then and adds a multi choice question "What kind of mobile do you have?" with four option. Three options can be ticked and in one option the user is able to type anything.

3.3 SECOND QUESTION
For question page two, the user creates the question "How satisfied are you with your mobile phone. Rank the following three options with 5 being 'very satisfied' and 1 being 'not satisfied at all'. The three options to rank are "Functionality", "Layout" and "Battery time". The user can then tick his ranking in a question matrix with five columns (1-5) and three rows (the options).

3.4 THIRD QUESTION
For question page three, the user creates the question "What is most important for you?" with the question instructions: "Rate the following three options with points. You have a total of 10 points. More points give more weight to your choice". The user proceeds to give three options "Send SMS and Email", "Spela" and "Surf on the internet" with a total of 10 points to distribute over the three options. A maximum of 10 points and a minimum of 0 points can be given to one question.
3.5 GIVE LAYOUT TO QUESTIONNAIRE
The user chooses a layout for the questionnaire from the available templates. He/She then chooses customizations for the layout, such as having each question on its own page.

3.6 QUESTION PREVIEW
The user previews each question on each page to see that the content of the questions and the question layout corresponds to his/her wishes.

3.7 ANSWER QUESTIONNAIRE
The user can finally choose to test answering his questionnaire. The user is guided through each page with a forward button and is able to tick and write answers.

3.8 CREATE URL FOR QUESTIONNAIRE Once the questionnaire is built, the user can create a URL for the questionnaire, copy the URL to the clipboard and then verify the URL in Internet Explorer in order to see if the questionnaire is displayed correctly on the web.
## D.6 Test specification for use case 3

<table>
<thead>
<tr>
<th>Preconditions 1.0</th>
<th>Create a questionnaire</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Click on ‘New’</td>
</tr>
<tr>
<td>1.2</td>
<td>Choose ‘questionnaire’</td>
</tr>
<tr>
<td>1.3</td>
<td>Wait until window ‘New questionnaire’ opens up</td>
</tr>
<tr>
<td>1.4</td>
<td>Type name ‘MyTestQuestionnaire’ in field ‘questionnaire name’</td>
</tr>
<tr>
<td>1.5</td>
<td>Click on option ‘Enable in some mode’</td>
</tr>
<tr>
<td>1.6</td>
<td>Click ‘Next’</td>
</tr>
<tr>
<td>1.7</td>
<td>Click ‘Finish’</td>
</tr>
</tbody>
</table>

### 2.0 First Question

<table>
<thead>
<tr>
<th>Step</th>
<th>Test step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>RightClick on choice 'Section' on left side menu</td>
<td>The option 'New Section' appears</td>
</tr>
<tr>
<td>2.1.1</td>
<td>If Section is marked, rightClick on marked 'Section'</td>
<td>The option 'New Section' appears</td>
</tr>
<tr>
<td>2.2</td>
<td>Click on ‘New Section’</td>
<td>Section Properties window appears</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Wait until 'Section Properties window' appears, then type 'Mobil' in field 'Section caption'</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Click on ‘OK’</td>
<td>Choice ‘Mobil’ appears under ‘Section’ in right hand menu</td>
</tr>
<tr>
<td>2.5</td>
<td>RightClick on choice ‘Mobil’ on left side menu</td>
<td>Choice ‘New question’ appears</td>
</tr>
<tr>
<td>2.6</td>
<td>Click on ‘New question’</td>
<td>Select type of question’ window appears</td>
</tr>
</tbody>
</table>

#### In ‘Select type of question’ window

<table>
<thead>
<tr>
<th>Step</th>
<th>Test step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>Choose ‘Multi options’</td>
<td>Properties of Multi Option Question’ window appears</td>
</tr>
<tr>
<td>2.8</td>
<td>Click on ‘OK’</td>
<td></td>
</tr>
</tbody>
</table>

#### In ‘Properties of Multi Option Question’

<table>
<thead>
<tr>
<th>Step</th>
<th>Test step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9</td>
<td>Write ‘Har du en mobilt?’ in field ‘Question caption’</td>
<td></td>
</tr>
<tr>
<td>2.10</td>
<td>Click on ‘+’ to add new options to ‘Caption’ field</td>
<td></td>
</tr>
<tr>
<td>2.11</td>
<td>Type: Ja</td>
<td></td>
</tr>
<tr>
<td>2.12</td>
<td>Click on ‘+’ to add new options to ‘Caption’ field</td>
<td></td>
</tr>
<tr>
<td>2.13</td>
<td>Type: Nej</td>
<td></td>
</tr>
</tbody>
</table>
2.14 Click on 'Ok' in right side menu. 'Properties of Multi Option Question' window closes.

2.15 Click on 'New question' in 'Select type of question' window. Choice appears.

2.16 Click on 'Multi option' in 'Select type of question' window. "Properties of Multi Option Question" window opens.

2.18 Click on OK in "Properties of Multi Option Question" window.

2.19 Click on 'Question caption' field.

2.20 Type: 'Vilken Mobil har du?'

2.21 Choose 'Answer required'.

2.22 Click on '1' to add first choice to caption field.

2.23 Click on 'Caption' field and type 'Samsung' in it, then click on empty space.

2.24 Click on '1' to add second choice to caption field.

2.25 Click on 'Caption' field and type 'Apple' in it, then click on empty space.

2.26 Click on '1' to add third choice to caption field.

2.27 Click on 'Caption' field and type 'HTC' in it, then click on empty space.

2.28 Click on '1' to add fourth choice to caption field.

2.29 Click on 'Caption' field and type 'annat' in it, then click on empty space.

2.30 Double click on 'annat' in Option Properties' window.

2.31 Click 'Use a textbox control (string)'.

2.32 Click on OK. Goes back to "Properties of Multi Option Question".

2.33 Click on OK in "Properties of Multi Option Question".

3.0 **Second Question**

Right click on 'Vilken mobil vill du ha?' on left side menu. "New section" choice comes up in Section Properties window.

3.1 Choose new section. "New section" choice appears.

3.2 Write 'Hur nåjd är du?' in Section caption field.

3.3 "Hur nåjd är du?" choice appears on right hand side menu.

3.4 Click on OK.
3.5 Right click on "Hur nöjd är du" choice in left side menu. "New question" choice appears
3.6 Click on "New question". "Select type of question" window appears
3.7 Click on "Matrix" choice.
3.8 Click on 'Single choice' option.
3.9 Click on Rows and type 3.
3.10 Click on Columns and type 5.
3.11 Click on OK. "Properties of Matrix question" window appears

In "Properties of Matrix question" window
3.12 Click on Column 1 and give it name 1.
3.13 Repeat 1.52 for all other Columns.
3.14 Click on first row and give it name.
3.15 "Funktionalitet".
3.16 Click on second row and give it name.
3.17 "Utseende".
3.18 Click on third row and give it name.
3.19 "Battertid".
3.20 Click on "Matrix caption" and type "Kryssa hur nöjd du är med din mobil. 5 betyder jätte nöjd och 3 betyder inte alls nöjd".
3.21 Click on OK. "Properties of Matrix question" window closes.

4.0 Third Question
4.1 Right click on "Hur nöjd är du" choice on left side menu. "New section" choice comes up
4.2 Click on "New section". Section Properties window appears
4.3 Click on "Section caption".
4.4 Type: "Vad tycker du är viktigast?"
4.5 Click on Ok. Right click on: Vad tycker du är viktigast in right side menu.
4.6 Click on: Question caption.
4.7 Write: fördela totalt 10 poäng på följande
4.8 3 aspekter. Flera poäng ger mer vikt.
4.9 Click on '1' and add first choice.
4.10 Type: Skicka sms och epost.
4.11 Click on '2' and add second choice.
4.12 Type: Spela
4.13 Click on '3' and add third choice.
4.14 Type: Surfa på nätet.
4.15 Click on: Required to rate all items
4.16 Click on: Limit of points per item
4.17 Type "0" in Minimum field
4.18 Type "10" in Maximum field
4.19 Click on Apply
4.20 Click on OK  "Properties of Rating Question"

5.0 Give Layout to Questionnaire
5.1 Click on Layout tab in bottom menu  Layout folder in right side menu
5.2 RightClick on Layout folder  appears
5.3 Click on: New layout  Choice New layout appears
5.4 Click on: Style  Pop-up window appears
5.5 Choose: Template
5.6 Click on: One page per section  Page Preview is initialized
5.7 Click on: Automatically save after each page
5.8 Click on: OK

6.0 Questionnaire Preview
6.0 Check if Preview is automatically initialized and wait for "Welcome to This"
6.1 Click on Page 2 in left side menu  Option Mobile appears
6.1.1 If Option Mobile is already visible  First question page "Mobile?"
6.2 Click on "Mobil" option
6.3 Wait until "Welcome" page goes away  appears
6.4 Click on Page 3 in left side menu  Option "Hur nåjd är du?" appears
6.4.1 If Option Mobile is already visible  Second question page "Hur nåjd är du?" appears
6.5 Click on option "Hur nåjd är du?"
6.6 Wait until First question page goes away
6.7 Click on page 4 in left side menu  Option "Vad tycker du?" appears
6.7.1 If option "Vad tycker du?" is already visible  Third question page "Vad tycker du ar viktiga?" appears
6.8 Click on option "Vad tycker du?"
6.9 Wait until second question page goes away
6.10 Click on page 5 in left side menu  "Thank you for this survey" page appears
6.11 Wait until "Vad tycker du ar viktiga?" page goes away

7.0 Answer Questionnaire
7.1 Click on "Play" button  Pop-up window appears
7.2 Wait until the "Forward" button in the pop-up window is visible
7.3  Click on Forward button  
7.4  Wait until next page is visible  
7.5  Answer first question  
7.6  Answer second question  
    Pop-up window will change to next page  
7.7  Click on Forward button  
7.8  Answer next question  
7.9  Click on Forward button  
7.10 Answer last question  
    Pop-up window will change to next page  
7.11 Click on Forward button  
    RightClick on NyTestQuestionnaire tab in upper menu  
7.12 Click on close  
    Wait for Close confirm message to appear  
7.14 Click on OK
D.7  Use case 4: Create, count and pass SQL-selection to DecideIT tree for calculations

**Brief Description**

Use case 4, the user would like to decide between three test automation tools. He/She uses statistics from a database to make an informed decision. Selection criteria for the first selection are amongst other tester experiences over three years. A second selection highlights the three most used testing tools. Those selections are then combined and used as weights in a decision tree calculation in order to find the right alternative.

**Actors**

Primary actor: decision analyst; Robot Framework
Secondary actor:

**Basic Flow**

4.1 CREATE AND COUNT A NEW SELECTION
This use case starts by creating a new selection. In this example use case, the user makes a first selection, choosing from all testers in Sweden, using open-source TestAutomation tools, with an experience over three years. From this first selection, a second selection groups together the three most used testing tools and calculates the percentage of usage for those three tools.

4.2 INCORPORATE OWN SELECTION INTO DECISION TREE
This use case uses the selection from use case 4, test case 1.1 and incorporates them as weights into calculations of an existing decision tree.

4.3 EXECUTE OWN SELECTION
Calculations for the decision tree are executed. Statistics from the database are part of the calculations. A screenshot of the result is taken.
## D.8 Test specification for use case 4

<table>
<thead>
<tr>
<th>Use Case 4: Create own selection, count it and execute own selection in dialog</th>
<th>Test case description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Create and build new dialog flow. Run use case 1, test case 1 and 2. After use case 1, test case 1 and 2 have run, begin with 2.0</td>
<td></td>
</tr>
</tbody>
</table>

### Preconditions 2.0

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Click on 'Add Selection' in DecisionIT. Wait for 'New Selection' Wizard Window comes up.</td>
</tr>
<tr>
<td>2.3</td>
<td>Click on 'Selection name' field</td>
</tr>
<tr>
<td>2.4</td>
<td>Type 'MyTestSelection'</td>
</tr>
<tr>
<td>2.6</td>
<td>Click on SQL field</td>
</tr>
<tr>
<td>2.6</td>
<td>Type SQL-code for selection</td>
</tr>
</tbody>
</table>

### 3.0 Run RIDE for-loop

In test step 2.6 of this use case, run for-loop which incorporates 8 different parameters into the SQL-statement for each for-loop iteration.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>Click on: Result</td>
</tr>
<tr>
<td>2.8</td>
<td>Wait for Result to appear</td>
</tr>
<tr>
<td>2.9</td>
<td>Click on 'Own value' field</td>
</tr>
<tr>
<td>2.10</td>
<td>Click on 'Own value name'</td>
</tr>
<tr>
<td>2.11</td>
<td>Type: nr_test_types</td>
</tr>
<tr>
<td>2.12</td>
<td>Click on scale field</td>
</tr>
<tr>
<td>2.13</td>
<td>Click on: Scale with interval</td>
</tr>
<tr>
<td>2.14</td>
<td>Type: 0-100 in interval</td>
</tr>
<tr>
<td>2.15</td>
<td>Click on 'Own value parameter'</td>
</tr>
<tr>
<td>2.16</td>
<td>Type 3</td>
</tr>
<tr>
<td>2.17</td>
<td>Wait for table with columns 'Own value' and 'nr_test_types' type: 100 for 3 test types; 60 for 2 test types and 20 for 1 test type</td>
</tr>
<tr>
<td>2.19</td>
<td>Click on: OK</td>
</tr>
<tr>
<td>2.20</td>
<td>Click on: Use SQL result as:</td>
</tr>
<tr>
<td>2.21</td>
<td>Choose: weight</td>
</tr>
<tr>
<td>2.22</td>
<td>Click on: Use Own value as:</td>
</tr>
<tr>
<td>2.23</td>
<td>Type: value</td>
</tr>
<tr>
<td>2.24</td>
<td>Click on: OK</td>
</tr>
<tr>
<td>2.25</td>
<td>Click on: Compute</td>
</tr>
<tr>
<td>2.26</td>
<td>Wait until DecisionIT has incorporated the calculations and built up a decision tree</td>
</tr>
</tbody>
</table>
D.9 Use case 5: Choose email template, give layout and test-send email

Brief Description

Use case 5 allows the user, to create an email from an existing template. The user can then customize the layout for the email. In this use case, the user gives the email a header, a body and inserts and an image. The user can then test sends the email to a google account and verifies if the email is displayed correctly in the web-browser.

Actors
Primary actor: user; Robot Framework
Secondary actor: gmail

Basic Flow

5.1 CHOOSE TEMPLATE FOR EMAIL
The user chooses an email template for the email. The template name is 'Blue Master Template'.

5.2 GIVE HEADER, BODY AND IMAGE TO EMARKETING EMAIL
The user chooses to write the header 'This is the Header, it will be dark blue and look fantastic'. The user then writes in the body 'The text will be in turquoise and mix well with the header’ and then chooses an image to be placed to the right of the email text.

5.3 TEST SEND AND VERIFY EMAIL IN WEB BROWSER
The user then writes an email address in to the email template and test sends the email. The user then goes into google chrome and verifies if the email has arrived and is displayed correctly.
D.10 Test specification for use case 5

<table>
<thead>
<tr>
<th>Use Case 5: Create, customize, test, send and verify email</th>
<th>Test case # and Test steps</th>
<th>Test case/step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions 1.0</td>
<td>Choose email template</td>
<td>The email will be send with blue layout</td>
<td></td>
</tr>
<tr>
<td>1.1 Click on “New”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2 Click on: “Message template”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3 Click on: Template name field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4 Type: MyTestEmail</td>
<td>If “OnlytestSend” is chosen, then change to “SendDrafts”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5 Click on: Next</td>
<td>Check if Email is chosen In Document</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6 Check if Email is chosen In Document</td>
<td>type: if not, choose it</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7 Click on: Basic template on master template</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8 Choose: Blue master template</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9 Click on: Finish</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Email template is chosen 2.0</td>
<td>Give layout to email</td>
<td>The email message has a header, body and image</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>If there exists an add option, hover and click on it</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>wait for: Header text field to show up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3 Click on: Header text field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4 type: This is the Header, it will be dark blue and look fantastic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5 Click on: Text field</td>
<td>type: The text will be in turquoise and mix well with the header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6 scroll to Right image area and click on ‘Select’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Wait for: ‘Select image’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>If option ‘Bu image’ exists, choose it, else choose ‘Dark Color image’</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.9 Click on: OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.10 wait for text: ‘This is the Header...’ to show up in main window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Email is written 3.0</td>
<td>Test send email marketing email and verify</td>
<td>The email message has arrived in an inbox and looks as planned</td>
<td></td>
</tr>
<tr>
<td>3.1 Click on Send To message field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2 type: <a href="mailto:michfieldbodovico@gmail.com">michfieldbodovico@gmail.com</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3 Click on: Subject field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4 type: Automated Test EmailMichelle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5 Click on “From” field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>type: <a href="mailto:RobotFrameworkAndSikuli@gmail.com">RobotFrameworkAndSikuli@gmail.com</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>click on: save Email Template is saved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Click on: Play If 'Test send outwards' window shows,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>click on send</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3.9.1</strong></td>
<td>If error message shows, click on:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Close window If 'Test send outwards' window is still</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>open, close it</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.12</td>
<td>Click on Google Chrome icon Google Chrome opens</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.13</td>
<td>In gmail Inbox Find the google inbox icon and click on it</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>In gmail inbox click on refresh button</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>wait for RFandSikuli mail to show up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>Click on email</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Take screenshot If one can read 'This is a header' text in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>email, close chrome Back to AUT If one cannot read 'This is a header' text,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.19</td>
<td>try to expand email; if that is not possible, close chrome Back to AUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>if the Robotframework window is open, make it smaller Back to AUT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D.11 Use case 6: Verify report in Report Portal

**Brief Description**

Use case 6 allows the user to view, save and archive a report for calculations made on a decision tree. The user opens the report for the right decision tree, saves it as a JPEG-file and then archives the report. Use case 6 requires the test cases 1.1 to 1.3 from use case 1 to be executed in order to be able to view a report.

**Actors**

Primary actor: decision analyst; Robot Framework
Secondary actor: gmail

**Basic Flow**

6.1 CREATE NEW DECISION TREE
This use case starts a decision tree framework. The user clicks on “New” and chooses the option “Decision Tree”, validates the number of nodes and confirms

6.2 BUILD UP DECISION TREE
This use case starts with a framework of a decision tree in the view field. The user names the tree and the two main options. For the first main option, the user creates an event with three nodes. Afterwards the user gives the nodes names, probabilities and values within two intervals.

6.3 EXECUTE DECIDE IT
This use case starts when the example decision tree is built up with two main nodes and an event on one of the main nodes. The event has three nodes which have been defined in 1.2. The user chooses to calculate a cumulative profile, a risk profile and a value tornado for the decision tree.

6.4 Verify Report in Report Portal
D.12  Test specification for use case 6

<table>
<thead>
<tr>
<th>Use Case 6: Test case # and Test</th>
<th>Test case/step description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions: Use case 1 1.0-3.0</td>
<td>Run use case 1, test case 1-3</td>
<td>Calculations on &quot;UseAutomation?&quot; decision tree are executed</td>
</tr>
<tr>
<td>Calculations on &quot;UseAutomation?&quot; decision tree have been executed 4.0</td>
<td>Verify execution report in a report portal</td>
<td></td>
</tr>
</tbody>
</table>

Click on 'Report Portal' button in Decide IT
Wait until Reports Overview comes up in a new browser window
Click on 'Refresh'
Click on Run 'Newest Decision Tree Execution Report'
Wait for 'Newest Decision Tree Execution Report [run]' window back to Newest Decision Tree Execution Report
Click on 'Which tree?' tab and select 'UseAutomation?' back to Newest Decision Tree Execution Report
Click on 'Format' tab
Choose 'JPEG' format
Click 'Run Report'
Execution Report to come up back to Newest Decision Tree Execution Report
Click on 'File'
Click on 'Save'
Save report in default folder Open folder and verify that report is there
Close folder
Go to 'Archiving' tab
Click on 'Archive report'
Click on 'View the archived report'
Wait for 'UseAutomation Decision Tree Execution Report archived' to come up back to Decide IT
Close Report Portal Back to Decide IT
Appendix E

Code from the library 'myCommandsMiluPC' and 'commonFunctions'

Figure E.1: Code for library 'myCommandsMiluPC' part 1

Figure E.2: Code for library 'myCommandsMiluPC' part 2
Figure E.3: Example of the Python file 'commonFunctions’ with code that has been broken out of test scripts
Appendix F

RIDE reports and logs

This is a picture of a successful test report. An unsuccessful test report is color-marked in red.

Table F.1: A successful test report from RIDE

The following is a more detailed picture which shows documentation of a test case within a successful test suite in RIDE.

Table F.2: Details from a test case in a successful test report from RIDE
APPENDIX F. RIDE REPORTS AND LOGS

The logs in RIDE are even more detailed and show each action which is performed. The next image shows a detailed test execution log.

Table F.3: A Test Execution Log

This execution log can be opened to show execution details.

Table F.4: Details of a Test Execution Log
Appendix G

Test Statistics

<table>
<thead>
<tr>
<th>Use case</th>
<th>Nr of cases</th>
<th>Nr of test steps</th>
<th>Average nr. of test runs until stable</th>
<th>Average test execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case 1</td>
<td>4</td>
<td>56</td>
<td>84</td>
<td>3.0 min</td>
</tr>
<tr>
<td>Use case 2</td>
<td>8</td>
<td>264</td>
<td>65</td>
<td>9.8 min</td>
</tr>
<tr>
<td>Use case 3</td>
<td>4</td>
<td>120</td>
<td>63</td>
<td>9.8 min</td>
</tr>
<tr>
<td>Use case 4</td>
<td>3</td>
<td>86</td>
<td>25</td>
<td>2.6 min</td>
</tr>
<tr>
<td>Use case 5</td>
<td>3</td>
<td>47</td>
<td>67</td>
<td>2.2 min</td>
</tr>
<tr>
<td>Use case 6</td>
<td>4</td>
<td>22</td>
<td>64</td>
<td>4.3 min</td>
</tr>
</tbody>
</table>

Table G.1: Test Statistics

Total average time to run whole test suit: 48.29 min
Total lines of code in the test suite (+cleanUp scripts): 1949
Average use case execution time: 5.28 min
Loading reports or previews in AUT: 1 to 35 seconds;
Nr of reports loaded in AUT: 7
Execution of an operation in AUT: ~48s;
Nr of operating executions in AUT: 12
Deleting in AUT takes: ~33.4s;
Nr of deletions in AUT: 6
Appendix H

RIDE menu and BuiltIn library

RIDE offers "play", "stop", "pause", "continue" and "step over" buttons.

![RIDE menu](image)

**Figure H.1:** RIDE menu

Besides those buttons, the vast "BuiltIn" library of RIDE offers many additional possibilities such as running/stepping over a test case or keyword, if a certain condition is fulfilled and test suites can therefore be adjusted to exceptions, possible errors, etc..

![RIDE BuiltIn library](image)

**Figure H.2:** RIDE BuiltIn library