Thomas Gruber, BSc.

A Robot Framework Library for Automated GUI Testing using the Ranorex API

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Softwareentwicklung-Wirtschaft

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl-Ing. Dr.techn. Eugen Brenner

Institute of Technical Informatics
Head: Univ.-Prof. Dipl-Inform. Dr.sc.ETH. Kay Uwe Römer

Graz, October 2018

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

The Robot Framework is an open-source test automation framework for keyword-driven acceptance testing that builds upon a simple, text-based syntax for defining test cases. It uses external libraries for providing actual test functionality. This thesis describes the RanorexLibrary and its implementation, a library that provides the capabilities of the Ranorex API to the Robot Framework. The proprietary tool Ranorex enables the tester to automate graphical user interfaces for desktop, web, and mobile applications by abstracting the underlying user interface technology completely. The end product is a way of writing Robot tests that have the same look and feel independent of the underlying technology and provide a clean interface to any GUI. Without the RanorexLibrary, each UI technology had its own library in the Robot Framework with its own keywords that were completely incompatible (if the technology is supported at all). A technical challenge to overcome in this thesis was the fact that Robot is built as a Python script, but Ranorex is a set of .NET libraries, so IronPython was used as intermediate layer to make them work together.

Contents

List of Figures

1 Introduction

> Like all good books, and possibly a few bad ones, this one starts with an introduction.

This is how *IronPython in Action* ([1]), an awesome book about the IronPython language/framework, starts its introduction, and I have decided to do the same, since IronPython provides the connection that is necessary to make the project that the RanorexLibrary is possible in the first place.

This introduction will cover a few basic questions about this thesis. The most important questions being "What is the RanorexLibrary?", "Who needs this RanorexLibrary?", and "How can the RanorexLibrary be used to automatically test Graphical User Interfaces?". The introduction will present the problem that needs to be solved and also how the RanorexLibrary can fill this gap.

1.1 The Issue to Solve

There are many test automation solutions and frameworks on the market, starting from open source tools and ranging to expensive Enterprise all-in-one solutions. When establishing test automation for a project, the question of which tool to use is maybe one of the most important questions to answer in this process, especially since this decision often creates a vendor lock-in and changing to another tool very often means to rebuild the whole test automation infrastructure from scratch.

Depending on the exact automation requirements, different licenses are dominant: Unit testing frameworks are very often open source tools.[1] Frameworks for Graphical User Interface testing on the other hand are often complex proprietary tools[2] that often come with sophisticated additional functionality like test case management or reporting.

Ranorex (presented in more detail in section 4.2) is a GUI test automation tool that offers advanced object recognition across many technologies. It abstracts the underlying UI technology and makes a test for a Java Swing application look exactly the same as a test for a WPF application. Within Ranorex, a click action abstracts the underlying control and always is just a click, which makes the Ranorex core technology very flexible and powerful. However, Ranorex is a proprietary tool and offers many features, most of them not well-portable to other platforms or frameworks. Another drawback is that Ranorex was never implemented with the idea to create a keyword-driven test framework, thus it lacks many of the features that are standard in this area.

On the other hand, there is the Robot Test Automation Framework (explained in more detail in section 4.1), an open source, keyword-driven test automation tool that builds upon Python. It works cross-platform, is interpreted instead of compiled, and is easily extensible through small scripts and libraries. Its strength is a very simple, text-based notation that builds upon keywords. Every possible action is modeled by a keyword, and it is easy to create new keywords from other, more basic keywords. It uses external libraries for defining the semantics of a keyword. And although there are a few libraries that enable Robot to perform GUI testing, they mostly focus on one specific technology (like Java Swing) and lack the overall power of Ranorex.

What both these tools have in common is their approach of making test automation easier to use for non-developers. Both don't require coding skills, and abstract actions as atomic modules (`keywords` in Robot and `actions` in Ranorex). Both can create more complex actions from basic actions (`higher`

[1] Wikipedia lists hundreds of them, and most use an open source license: `https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`

[2] Overview over GUI testing tools in the Wikipedia: `https://en.wikipedia.org/wiki/Comparison_of_GUI_testing_tools`

`order keywords` in Robot and `modules` and `module groups` in Ranorex) and both use similarly stuctured test suites.

However, there is currently no well-maintained and powerful integration of the Ranorex API into the Robot Framework. This integration would enable a tester to use keywords within the Robot Framework that trigger Ranorex GUI tests. Currently, for doing UI testing, a Robot user has to choose the correct library (e.g. the SeleniumLibrary for web testing or the SwingLibrary for Java Swing applications), if there is a library for this technology in the first place. Automating a GUI independent of its underlying technology is currently not possible. A clear benefit of the RanorexLibrary would be to only have one single notation for all technologies, meaning it is not necessary to learn another library for another technology, and to make end-to-end testing simple and natural. In Ranorex, adding a file to a net drive on the desktop and validating its existance on the corresponding web application or a mobile app is seamless, while without Ranorex it would be hard and require many different libraries' keyword sets.[3]

1.2 The Aim of the RanorexLibrary

The goal of this thesis is to analyze the options to integrate the Ranorex functionality into the Robot Framework, and then to implement such a solution that should have the name "RanorexLibrary".

First, basic questions have to be answered, like: "Is such an integration possible in the first place?" and "How could this integration look from a technological standpoint?". Since Ranorex builds upon .NET and Robot is based on Python, they do not seem to be the perfect match for each other. Thus, first a proof of concept has to be established, only then can the real functionality be implemented into specific keywords.

[3]Is it "Run Application" or "Start Application" or "Start Program"? When using different libraries, the same functionality may be offered by different keywords, and this could make a Robot test hard to write and maintain, and very unlogical. Clicking an element could require the keyword "Click" in one line, but the keyword "Click Element" in the next. Although it is possible to unify those keywords by using higher order keywords, this still has to be done by the end users themselves, and it is not an elegant solution.

Ranorex offers many features apart from just object recognition. Another question to answer is, how many of these features can (and should) be translated to Robot. While the Ranorex repository might seem like it could be integrated well and easily[4], the Ranorex report should probably not be integrated as Robot provides its own independent report format.

Testing web applications is one area where Robot is very strong due to an integration of the Selenium framework, which in itself is the standard web automation framework and is supported by all major browsers today. The RanorexLibrary will also provide web testing capabilities, but when testing only web applications within Robot, the SeleniumLibrary might be the better solution as it doesn't require Ranorex licenses. For mobile app testing, the situation is similar. With Appium, there is a strong and open source solution to do so. The RanorexLibrary should support both web and mobile app testing in addition to desktop testing to really provide the full benefit of the Ranorex core.

The RanorexLibrary should be used in real applications, and to enable end users to incorporate Ranorex into their test project, it is necessary to have a good documentation, guidance and tutorials. Thus, writing these materials is also considered part of the RanorexLibrary, as the library itself would not be useful without this extra content.

1.3 Methodology

This thesis has two destict parts: One is the RanorexLibrary itself (containing the actual code itself, documentation, tutorials etc.) and the other one is the theory part that is represented by this text.

The practical part, represented by the RanorexLibrary, is a small software project. It is written as a Python script and hosted on GitHub. This Python script slots into the Robot Framework infrastructure at the *library* position as presented in section 4.1.2. It uses IronPython functionality and can only be run using the IronPython environment. Calls to the Ranorex API

[4] In the end it turned out that this was a wrong assumption, as explained in section 5.2.2.

are therefore *native* and don't require any type conversions or specific prerequisites.

The theory part should put the RanorexLibrary into the correct perspective within the test automation scope and describe its functionality. Much of the presented information is based on resources from the Internet (GitHub project Pages, official web resources of projects or companies, etc.) or scientific material like journal articles and books. The former are usually referenced by giving the corresponding URLs in a footnote, while the latter are referenced in IEEE style.

1.4 Content of this Thesis

This section provides a short overview over the thesis. It should enable the reader to quickly find specific information.

Chapter 2 gives a short introduction to other, already-existing integrations of the Ranorex API into the Robot Framework. It also contains a discussion about why these integrations don't fulfill the full requirements set and why this thesis still makes sense, even though other integrations exist.

Chapter 3 is a more theoretical part that describes all the important terms related to software testing and test automation that build the foundation for this thesis. There, for all these terms there is a definition of how these terms are used within this thesis in order to build a common ground between author and reader.

Chapter 4 describes all the frameworks that are used in the RanorexLibrary. It contains a detailed overview over Ranorex, the Robot Framework and IronPython, as well as a sample integration that uses Selenium instead of Ranorex in order to see how a similar, already-existing integration works.

Chapter 5 lists the requirements for the RanorexLibrary's implementation. It lists what the goals and acceptance criteria are, and what should be possible when using the RanorexLibrary.

Chapter 6 gives implementation detail for the RanorexLibrary. It is a loose documentation for how the Library works internally and also lists the important keyword sets and how they are implemented, making extending this list easy.

Chapter 7 gives all the information that is needed to set up a working test suite using the RanorexLibrary. It describes all the necessary steps to set up the environment, and also gives a small example solution. This chapter also contains a small set of best practices that immensely help when creating robust and maintainable tests using the RanorexLibrary.

2 Already-Existing Integrations

There already are a few existing solutions that integrate the Ranorex API into the Robot Framework. Their existance, however, does not mean that the underlying problem has already been fully solved. All of these already-existing integrations have some problems that make them insufficient to meet the requirements.

2.1 ranorex-robot-library

This project (hosted on Google Code[1]) was started in May 2013. It is linked from a Ranorex Forum entry[2] and was started by a Ranorex community member. However, the project hasn't been updated lately, and links to the downloads on the Google Code project page lead to 401-errors. The Google Code project itself has been shut down in early 2016[3], and the source code of this library can't be downloaded from there anymore. Therefore, this integration can be considered dead and inaccessable.

2.2 robotframework-RanorexLibrary

This project is much more mature than the afore-mentioned one and very similar to the RanorexLibrary presented in this thesis. The project

[1]The project page can be found here: `https://code.google.com/archive/p/ranorex-robot-library/`

[2]Forum entry can be found here: `https://www.ranorex.com/forum/ranorex-integration-with-robot-framework-t1952.html#p20493`

[3]According to Google itself, there is only an archive available: `https://code.google.com/archive`

is hosted on GitHub[4]. It is published using the MIT license. Its main file, `rxconnector.py`, has a similar structure to how it is solved in the RanorexLibrary created in this thesis, however, the selection of implemented keywords is different (and doesn't reflect the Ranorex actions well): it doesn't implement all keyword parameters that Ranorex could support, and it works on the `element`-level within the Ranorex API, while the RanorexLibrary from this thesis works directly with `Unkown`-adapters[5]. Using adapters instead of elements has a few advantages: It is easier and more elegant, and it eliminates complexity overall. Using the adapter also puts the keywords closer to actual Ranorex API functions. Additionally, the two contributors to this project have both not been active on GitHub for the last years, and the last commit to the project dates from Jan 19th 2015, making the whole project only active for little more than half a year.

[4]Project can be found here: `https://github.com/alans09/robotframework-RanorexLibrary`

[5]Ranorex internally also uses the `Unkown`-adapter for performance reasons.

3 Terminology

This chapter aims to identify the place that the RobotLibrary occupies within the software test environment. The Robot Library is an *acceptance testing framework*, as explained in on the Robot Test Framework homepage[1], as well as used as a *test-driven development* tool. The approach that the Robot Framework takes is *keyword-driven testing*.

Ranorex[2] is a proprietary tool for Graphical User Interface (GUI) test automation. It supports testing on desktop, web and mobile platforms and is based on the Microsoft .NET framework.

All these terms that are relevant for the RanorexLibrary within the software testing scope are explained in more detail in this chapter.

3.1 Software Testing

What is the definition of software testing that this thesis builds upon? And which imiplications does this definition have for the RanorexLibrary? The definition, found in [2], that this thesis sticks to, is the following:

> Testing is the process of executing a program with the intent of finding errors.

This definition has a few interesting implications: First, in order to test an application, it is necessary to execute it. This distinguishes testing from other code quality processes like static analysis. The combination of this need and the typically longer execution times of GUI tests lead to a probably

[1]Link to the official Robot web presence: `http://robotframework.org`
[2]Official Ranorex web page: `https://www.ranorex.com`

surprising fact: GUI test automation takes by large magnitudes longer than unit tests or integration tests, and the execution time can go up into the range of days for larger applications.

Additionally, since testing has the goal of "finding errors", it is also crucial to have good tests that tend to find errors if something breaks for a next release of the system under test. In [2], the authors suggest to call a test *successful* if it finds an error, and *unsuccessful* if it fails to do so. In regression testing, this seems to shift slightly, however.

3.1.1 The Testing Pyramid

The testing pyramid (as shown in figure 3.1 on page 11) is a common concept in agile software development. It puts the different testing strategies into a common concept by aligning them on a triangle. On the bottom, there are the tests that should make up the main part of the testing efforts as they are cheap and fast in execution. On the top there should only be a few tests as they tend to be expensive and have a longer runtime.

Typically, on the bottom there are *unit tests*. They are often written by the developers themselves together with the actual code (*white box testing*). The middle step is sometimes split further, but most of the time it is referred to as *integration tests*. On this layer, the framework is tested in an input/output-manner (black box testing). On the top, there are the *end-to-end tests*, often also called *UI tests* or *acceptance tests*. These are tests that often actually mimic the user interactions with the application, often by clicking buttons, typing on the keyboard etc.

3.2 Agile Software Development

Agile software development is not a strict concept, but an anti-thesis to the waterfall model. There are many definitions of what *agile* means exactly. However, all the definitions are built upon the *Agile Manifesto*[3].

[3]The Manifesto consists of only 12 guiding rules, but still defines a widely-used software development process. The rules can be found all over the internet, for example in [3].

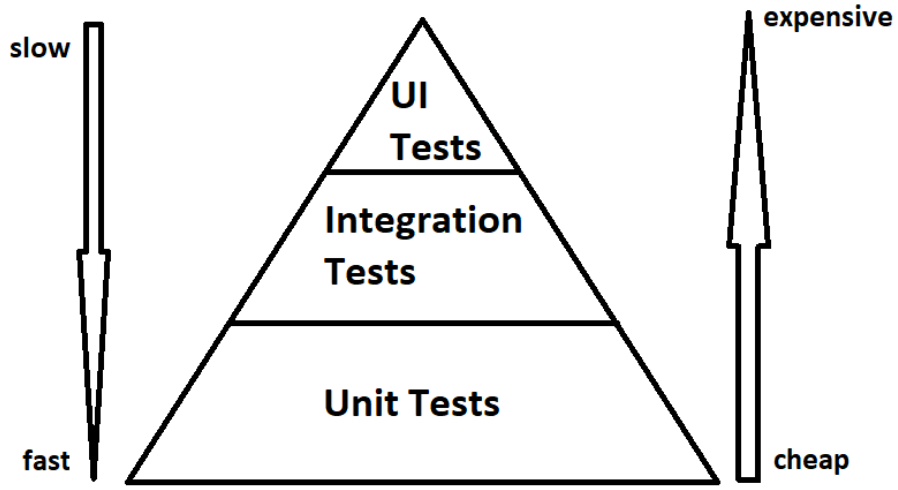


Figure 3.1: This image shows the testing pyramid as it is found in many sources about test automation. The middle layer is often divided even further, and the top layer sometimes has the name *Acceptance Testing* instead of *(G)UI Testing*.

The guiding idea behind agile is to put people into the center of the development process, not the process itself. Agile means to satisfy the customer needs quickly and early. It means to give developers the freedom and the power of decision to make many small, good iterations that increasingly satisfy the customer needs.

One common effect of agile development is that release cycles get shorter and that the releases themselves are smaller. This, in turn, means that testing becomes more critical in time. If the application still has to be tested for every release, test automation becomes increasingly important.

3.2.1 Test-driven Development

A very common approach in agile software development is *test-driven development*, a strategy where tests get written *before* the actual code that implements it. [4] gives a more detailed definition and introduction to the

topic, however, the guiding principle is to write a test first, then run the test (at this point it must fail since there is no actual code that implements the tested functionality, which in itself is a test for the test already), then to implement the actual code and finally to run the test again—and now that test should not report any error anymore.

Test-driven development is possible on every layer of the testing pyramid, most commonly used in unit testing, but also possible in UI testing, especially in combination with UI mockups. UI departments often create small UI applications that don't have any functionality yet, but already use the finished UI structure. Creating a test in this application before implementing the functionality can already be an acceptance test (see section 3.6 for more information).

3.3 Test Automation

Finding a definition for the term *Test Automation* is surprisingly difficult. A papaer making a literature review about this topic ([5]) fails to do so, and the ISTQB syllabus for the test automation engineer also loses no word about what test automation actually is. There seems to be the common knowledge that automated tests (in contrast to manual tests) are tests that are not run by a human, but by a system that is first trained or programmed by a tester.

Test automation is about defining a test in a way and within a framework, that the testing goal can be reached without human intervention. Both Robot and Ranorex identify themselves as test automation tools, and both fulfill these criteria: Test logic can be defined within their syntax and the tests can then be run by those systems without human supervision.

3.4 Graphical User Interface Testing

According to [6], a "GUI takes events (mouse clicks, selections, typing in textfields) as input from users, and then changes the state of its widgets".

There are many ways how an application can interact with the user: command line interfaces, audio interfaces etc. However, GUIs have developed to be a main way in consumer applications as they are perfectly suited for being shown on a screen.

To functionally test a user interface, it is necessary to interact with it in the standard interface that this GUI understands: usually mouse clicks and keybard inputs (and increasingly important via touches on a touch screen). If the GUI reacts in the specified way, the test can be considered ”green”, while it should be ”red” if the GUI does not react in the specified way.

3.5 Keyword-driven Testing

Pekka Laukkanen developed the foundation of the Robot Framework in [7]. There, the author constructed a language from scratch that had to fulfill specific requirements. Within that thesis, keyword-driven testing is defined as an extension to data-driven testing. In a first step, data is abstracted and moved away from the test itself. Most of the time, this data is given in an external file having its own format, often in a tabular form.

In the second step, not only data is moved away from the test itself, but also directives telling what to do. This adds another layer of abstraction between *how* a test step is implemented and *what* test step should happen. This layer of abstraction is built from keywords that move the technical detail away from the tester. Thus, a test automation engineer shoult be able to use simple directives (*keywords*) to construct tests without having to worry about the technical detail of *how* this keyword is interpreted.

A keyword’s interpretation (or implementation) can be moved away from the test automation engineer and could then be constructed and maintained by a developer. Most importantly, a change to the system under test does not require the test script to be changed, just the underlying keyword implementation, which makes more sense for higher level testing like acceptance testing.

3.6 Acceptance Testing

According to [8], acceptance testing is the act of assessing software with respect to requirements. Therefore, acceptance testing is the top level of the testing hierarchy (unit tests being on the other end). These tests are often constructed by the end user and not the developer or quality assurance team themselves. In the V-model of software development, the requirements are often depicted as the first step in the development process, and they are directly linked to acceptance tests. Acceptance tests have to be black-box tests since the internal system state is not relevant for the end user. Consequently, the end user has to interact with a system via an interface, often a graphical user interface. Therefore, acceptance testing is often done on the GUI level (although other approaches are possible, for example for APIs).

The Robot Framework describes itself as an acceptance test framework, and its implementation fulfills the requirements for an acceptance testing tool. It abstracts the actual test functionality away from the user (maybe even the customer) and offers them a clear and formal interface to the software by providing keywords, while this software doesn't even have to exist at this point yet.

4 The Frameworks in Use

This chapter deals with all the frameworks and applications that are used in the RanorexLibrary. Since the RanorexLibrary is an integration of the Ranorex API into the Robot Framework, these two external frameworks are presented. Robot allows external libraries to be used within its syntax; the RanorexLibrary is such a library.

Since Ranorex is strictly based on the .NET framework itself, and the Robot Framework builds upon Python directly, the obvious choice to make them work together is IronPython, a Python implementation in the .NET framework that allows the developer to use .dll files compiled in the .NET framework to be imported into Python applications.

Since the RanorexLibrary is a standard library for the Robot Framework, this chapter also takes a look at another example library that does something similar: The SeleniumLibrary. The SeleniumLibrary integrates Selenium functionality into the Robot Framework in order to enable the tester to access and automate web pages directly in the Robot syntax.

4.1 Robot Framework

> Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or

> Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases.[1]

The main source for this section about the Robot Framework is the introduction book [9], the official web page of the Robot project at [10], as well as resources that are linked from this site, for example the documentation or different libraries.

4.1.1 History of the Robot Framework

The earliest roots of the Robot Framework go back to [7], a Master's thesis dating to the year 2006 by Pekka Klärck (Pekka Laukkanen at the time of writing of his Thesis). In this thesis, the author developed the first concepts to his keyword-driven testing approach that are still the foundation for today's version of the Robot Framework.

In 2008, the author released his framework as open source project under the name *Robot Framework*, as stated on the author's homepage [11]. Today, the framework is available under the Apache License 2.0[2] and hosted on GitHub[3]. After being founded by the Nokia Networks[4] in its earlier days, Robot Framework today is fostered, developed and taken care of by the Robot Framework Foundation[5], a consortium of 23 companies that sponsor the whole infrastructure of the project.

4.1.2 Technical Detail

According to the official documentation[6], the Robot Framework consists of different layers. Figure 4.1 on page 17 shows this architecture as a flow

[1] Taken from `http://robotframework.org`

[2] Apache License: `http://www.apache.org/licenses/LICENSE-2.0.html`

[3] GitHub project page: `https://github.com/robotframework/robotframework`

[4] Nokia Networks web page: `https://networks.nokia.com`

[5] List of companies in this foundation: `http://robotframework.org/foundation/`

[6] Robot Framework documentation: `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html`

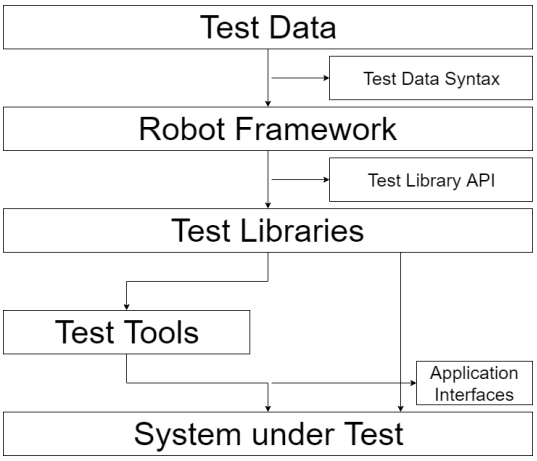


Figure 4.1: The architecture of the Robot Framework. Image drawn using the example of a similar image from the Robot user guide: `http://robotframework.org/robotframework/latest/images/architecture.png`.

chart, starting from the top layer of abstraction down to the actual system under test.

Test Data At the top there is the *Test Data* layer. Here, the tester provides input data to the Robot engine in form of data structured in the Robot test data syntax (that will be explained in section 4.1.3 on page 19). This data can be presented in files with one of the file extensions *.html*, *.xhtml*, *.htm*, *.tsv*, *.txt*, *.rst* or *.robot*. The data is given in a table-like format with different columns representing different settings. The plain text format has evolved to be the standard, and all examples will therefore be given using the plain text syntax.

Robot Framework The next layer is the Robot Framework itself. This layer is the bridge between the test data and the external libraries. It takes the input files and interprets and executes them. Robot itself internally is a set of Python scripts that can either run using plain Python[7], Jython[8] or

[7]Weg page of the Python project: `https://www.python.org`
[8]Web page of the Jython project: `http://www.jython.org`

IronPython[9]. This layer also takes care of things like reporting and logging. The unit the Robot Frameworks works on is the *keyword*, although Robot itself does not know the semantics of any keyword.

Test Libraries Robot itself provides the syntax, but actual keywords and their semantics are provided by external libraries. Those libraries are implemented either in Python or Java. Their main purpose is to provide a link to testing tools or the SUT itself by implementing keywords. The Robot Framework already has a few libraries that come as standard, for example the *BuiltIn*[10] library that provides basic keywords like `Should Be Equal`, `Log` or `Run Keyword If`, giving an interface for simple assertions, logging and conditional keyword execution. Other libraries that come as standard are `OperatingSystem` to access OS functionality, `String` to perform basic actions on strings and `Process` for running processes.

Test Tools Sometimes the test libraries themselves do not provide the actual testing functionality. Instead they only act as middleware between the Robot Framework and an external testing tool. The popular framework Selenium[11] for example provides the functionality to test web applications, the SeleniumLibrary (explained in section 4.4 on page 34) acts as a wrapper around it that provides keywords to the Robot Framework that can then be used within the Robot syntax. The RanorexLibrary itself is another example of a library that just links the Robot Framework to an external testing tool: Ranorex.

System under Test The tests performed by the Robot Framework are normally performed on an actual application under test. This application might be anything like an API, a library or a graphical user interface. If the whole technology stack works as expected, a keyword used in a Robot input file triggers an action in the SUT, thus automating it.

[9] Web page of the IronPython project: `http://ironpython.net`

[10] Documentation of the `BuiltIn` library: `http://robotframework.org/robotframework/latest/libraries/BuiltIn.html`

[11] Web page of the Selenium project: `https://www.seleniumhq.org`

4.1.3 The Robot Syntax

This section describes the Robot syntax. It is based on the official documentation[12] and illustrates the syntax with practical examples. Figure 4.2 on page 20 shows a small piece of example code.

File Formats and Directory Structure Test cases are created in a test case file that itself is automatically a test suite. Test suites can be nested into directories; a directory containing several test case files itself is a test suite containing test suites. A test suite directory itself can also contain other test suite directories, making a tree-like hierarchical structure of test suites possible. Test suite directories can have initialization files. In addition, there are resource files that define variables and user-defined keywords. The Robot Framework supports several file formats like HTML, TSV, plain text or reST. Robot can recognize files with one of these file extensions: *.html*, *.htm*, *.xhtml*, *.tsv*, *.txt*, *.rst*, *.rest*, and *.robot*. The examples in this thesis will always use the *.robot* extension to distinguish Robot files from all other files that may occur. Only the plain text format will be explained here, and all examples will also be given using the plain text format.

General File Overview A Robot file consists of tables. These tables each have a title, beginning with an asterisk character (*). By convention, most files use the format `*** Settings ***` though with three leading and three trailing asterisks, separated by a space from the table title. There are four possible table titles: `Settings`, `Variables`, `Keywords` and `Test Cases`.

Space-/Pipe-separated Format Robot tables consist of rows and columns. The rows are separated by newlines, the columns by either *two or more spaces* or a pipe symbol (|) enclosed by a space on both sides. Both formats can be mixed, as long as one line stays consistent. When using the pipe-separated format, leaving cells empy is simple, but when using the space separated

[12] Official Documentation of the Robot Framework: `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-syntax`

```
*** Settings ***
Documentation       This test should check if the Run and Close Application keywords work fine.
Library             RanorexLibrary      C:\\Program Files (x86)\\Ranorex 8.2\\Bin

*** Variables ***
${logMsg}                   Opened and closed application

*** Test Cases ***
Run The Calculator
    Log                     Try to run and close the Windows calculator.
    Run Application         calc.exe
    Close Application       /winapp[@packagename='Microsoft.WindowsCalculator']
    Log                     Opened and closed the Windows Calculator

Run The Ranorex Test Application
    Log                     Try to run and close the test application
    Run Application         C:\\test.exe
    Close Application       /form[@controlname='RxMainFrame']
    Log                     ${logMsg}
```

Figure 4.2: This listing shows some example code in the Robot Framework syntax in space separated plain text format. This file is a small test suite consisting of two tests. The screenshot is taken in VS Code with Robot syntax highlighting enabled.

format it is necessary to escape empty cells with either a backslash or the `${EMPTY}` variable.

Semantics How these lines are interpreted completely depends on the keyword that is used. Keywords may take arguments, but those arguments might be other keywords taking arguments again. To find out what a line does, it is necessary to know the semantics of the specified keyword. Please also note the indentation in figure 4.2 as it also has semantic meaning: More than one space is a table separator. Thus, the keywords themselves start in the second column, the first column being empty to specify that this line *belongs to* the test case above.

4.2 Ranorex

The Ranorex GmbH is the company based in Graz, Austria, that owns the product with the same name: Ranorex. The Ranorex GmbH as well as

the American subsidiary Ranorex Inc. belong to the Texan Idera Inc. since 2017[13]. After being founded in 2007 by a father and a son[14], the company grew rapidly over the next ten years with funding from the EOSS Industries Holding GmbH[15] in Graz. Today, Ranorex counts over 3500 customers in over 60 countries.

The main sources for the information in this chapter about Ranorex come from both the official web resources of the company found at [12] and the personal experience of the author as an employee of the company.

4.2.1 The Product Ranorex

Ranorex is a set of tools that enable its user to automate graphical user interfaces. The product is mainly geared towards test automation, although it is also possible to use Ranorex as process automation tool. A full Ranorex installation comes with several different tools. The two main applications are Ranorex Studio[16] and the Ranorex Spy[17].

Ranorex Studio is the main tool for creating test suites and test scripts. In its center there is the test suite that structures the test modules into a flow control. Ranorex Studio can be used to create these modules by either using the recording functionality, where Ranorex records the mouse and keyboard actions that a user performs, or by constructing tests manually using the smallest atomic unit that Ranorex Studio knows: the action. An action might be a mouse click on an element, opening or closing an application, a key sequence etc.

The Ranorex Spy on the other hand can be used to inspect the graphical user interface of all currently running applications in order to find identifiers for

[13]Article about the acquisition: `https://www.businesswire.com/news/home/20171023005356/en/Idera-Acquires-Ranorex-Doubling-Size-Test-Management`

[14]From the Ranorex Blog: `https://www.ranorex.com/blog/celebrating-10-years-of-ranorex-a-letter-from-our-ceo/`

[15]Web page of the EOSS Holding GmbH: `http://www.eoss.at`

[16]User Guide article to Ranorex Studio: `https://www.ranorex.com/help/latest/ranorex-studio-fundamentals/ranorex-studio/introduction/`

[17]User Guide article to Ranorex Spy: `https://www.ranorex.com/help/latest/ranorex-studio-advanced/ranorex-spy/introduction/`

the different UI elements. It can either be used to find a RanoreXPath for a given UI element, or for finding UI elements for a given RanoreXPath.

When working on pure code level, it is not necessary to use Ranorex Studio at all, since the API can also be accessed directly and Ranorex Studio itself just uses this API itself, however, using the Ranorex Spy is very important as this is the only way to find a RanoreXPath quickly and easily.

Ranorex also has some other tools that come with a full installation, like the Ranorex Recorder or several small helper tools like the Parallel Runner. Many of those provide important functionality that should also be mapped in the RanorexLibrary.

Licensing and Pricing

Ranorex is a tool under a proprietary license[18]. This means that it can't be used for free[19] (apart from a free trial version for 30 days). This section explains the licensing and pricing of the Ranorex licenses.

There are two types of licenses: Premium Licenses and Runtime Licenses. The former are needed to create and maintain tests, whereas the latter are sufficient for running tests only. The Premium licenses come in two flavors: Floating and Node Locked. All license types also come as Enterprise Licenses where the only difference to Premium Licenses is the level of support—the product is the same for all license types.

Premium Licenses Premium licenses are necessary for everything that involves creating and maintaining tests. When opening the Ranorex tools like Ranorex Spy or Ranorex Studio, a Premium license is necessary. The Node Locked License is cheaper, but it is bound to a physical machine. There are protection mechanisms in place that prevent this license to be used on Virtual Machines altogether. The license can only be transferred

[18]Information on pricing and licensing is based on the information on Ranorex' official pricing page: `https://www.ranorex.com/prices/`

[19]For trying out the RanorexLibrary for academic purposes, it is sufficient to contact the Ranorex sales team using `sales@ranorex.com` in order to get a time-limited trial license.

from one machine to another every 90 days. It is mainly meant for "an individual user working on multiple projects on a single physical machine". On the other hand there are Floating Licenses. They are installed on a License Server application within the local network of the license-owning company. The Ranorex tools can then be installed on as many (physical or virtual) machines as the Ranorex customer wants. As soon as an application like the Ranorex Studio is opened, it asks the server for a free license. If a license is found, the license is blocked and the application opens, if there is no free license, the end user gets a warning window and can't open the application. There always have to be at least as many licenses as users that want to use Ranorex concurrently.

Runtime Licenses If a Ranorex test is only run, a Runtime License is sufficient. Runtime licenses are always Floating Licenses. They work differently than Premium Licenses: Ranorex products like Ranorex Studio can't be opened with Runtime Licenses. However, if a test run is started, it also requires a valid license. A test might run using a Node Locked license (when run on the machine where the test is also created, blocking the mouse and keyboard), a Floating License or a Runtime License. The main difference with Runtime Licenses is that they are significantly cheaper than Premium Licenses. It is necessary to have as many free licenses as tests that should run concurrently.

4.2.2 The Three Layers of Ranorex

From a technical perspective, Ranorex is built in three different layers. The Ranorex sales team presents this layer structure to potential customers using [13]. On the lowest layer there is the pure .NET API, the next layer represents the mapping of real user interface elements to an internal representation of these elements, the Ranorex repository. And on the top layer, there is the Ranorex test suite that adds the real testing functionality, provides modularization, test runs and reporting.

```
public void customClick()
{
    Unknown button = new
        Unknown("/form[@controlname='RxMainFrame']/button[@controlname='RxButtonExit']");
    button.Click();
}
```

Figure 4.3: This is an example of a click performed with the Ranorex API. First, an element of the adapter-type `Unknown` is generated using the explicit RanoreXPath, then its method `Click()` is called.

Layer 1: The .NET API

This layer consists of the .NET API of Ranorex. It provides most of the functionality that deals with interacting with user interfaces and their automation. Typical functions that can be found in this API are functions like `Click()` or `GetAttributeValue()`. This API can be used without the need of the more advanced Ranorex products like Ranorex Studio. However, since on this layer the RanoreXPath is used for identifying objects (and the RanoreXPath can only be found easily using the Ranorex Spy tool), it is still necessary to access some of the Ranorex tools. The documentation of the API is publicly available and can be found on the internet.[20].

Layer 2: The Ranorex Object Repository

The next layer of Ranorex is the Ranorex repository. The layer represents an abstraction of the GUI of the system under test. Here, a user interface element like a button is represented in an internal data structure. Data objects within this data structure are usually referred to as *items*. These items have a name, a RanoreXPath with which the corresponding user interface element can be addressed and some meta information like a search timeout.

The main benefit of having this abstraction is that the third layer, the test suite, uses the repository items as work items. Thus, if the user interface changes in a new release, only the corresponding repository item needs to be changed once, independent of how often it is used in the test suite itself.

[20]Ranorex API documentation: `https://www.ranorex.com/Documentation/Ranorex/`

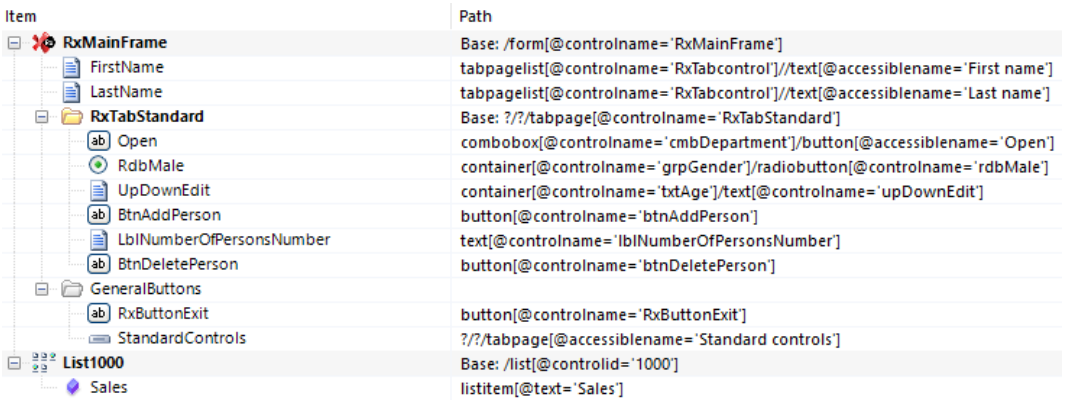


Figure 4.4: An example repository as displayed by the Ranorex Spy. It consists of root elements (`RxMainFrame` and `List1000`) and regular repository items as well as of Rooted Folders and Simple Folders to structure the items. All items consist of a name and a RanoreXPath (plus a few settings).

This makes maintaining the test easier in the long run, and is also at least a best practise in other automation frameworks that don't offer a repository function, as for example explained in [14]. This layer of abstraction decouples the actual system under test and the test logic. The more complex task—finding good and robust locators—can therefore be done by other poeple than creating the test logic.

Layer 3: The Ranorex Test Suite

On the third layer there is the test suite. The test suite itself uses the two layers below it and organizes it into modules and the modules into a flow control. On the highest level, there are test cases that run one after another, each itself consisting of recordings, the recordings in turn consisting of simple actions. All of these can be organized within a folder structure. On this level the binding of external data to internal variables happens (*data-driven testing*) and higher-level functionality as rerunning or looping test cases can be implemented.

With the test suites another very important part of testing comes: the reporting. Based on the test suite structure, Ranorex creates a report file for a test run. This report gives the tester the information they need by

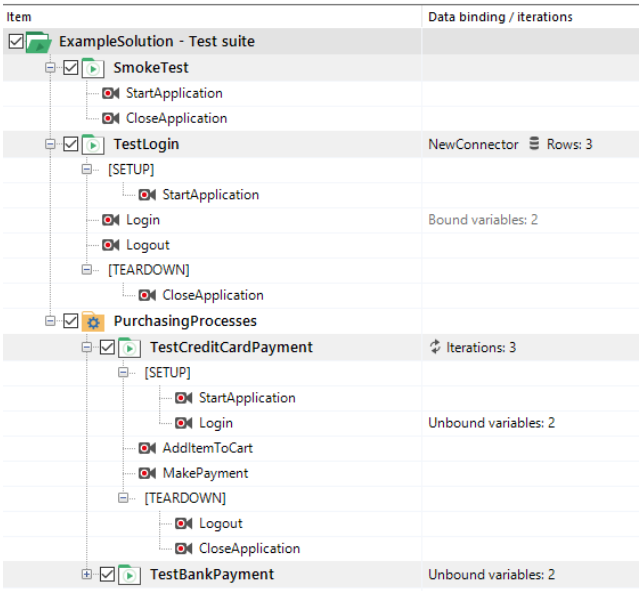


Figure 4.5: This figure shows a Ranorex test suite as it is shown by Ranorex Studio. It consists of Test Cases, Smart Folders and recording modules.

providing an overview of the whole test run as well as logging for every action that has happened in the test.

Building upon this layer, there might be another layer: *Test (case) management*. This layer is not part of the Ranorex product chain but has to be implemented additionally. Test management provides additional functionality like dashboards and charts. Test management creates the connection between several test runs, while Ranorex' functionality stops as soon as a test run is finished. Ranorex has a built-in support for a the TestRail[21] test management tool.

4.2.3 Object Recognition

Testing of Graphical User Interfaces is typically a black box testing approach. There is no knowledge about the internal application structure necessary

[21] Web page of the TestRail product: `www.gurock.com/testrail`

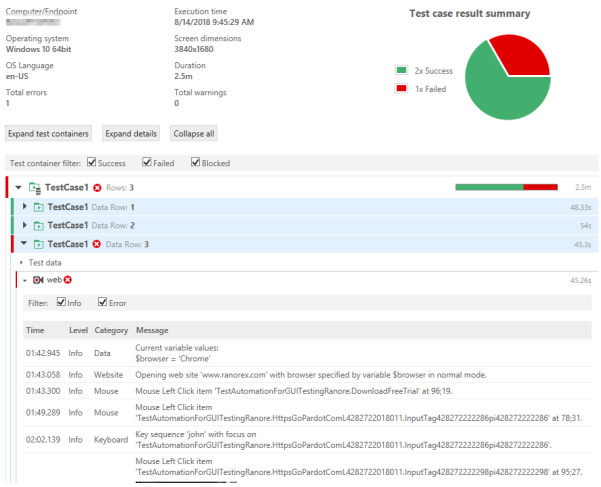


Figure 4.6: This figure shows an example Ranorex report file that is generated after each test run. It shows the success state of every test case and gives tracing information for every action that Ranorex has performed.

to perform the test. However, since common actions in GUI testing are interacting with specific UI elements (like buttons or text fields), a GUI testing tool needs a way to find those elements on the screen in the first place.

The easiest approach is to always interact with the same screen locations. For example, performing a click on the same pixel will result in the same interaction with the GUI. However, a test conducted in this way has some serious drawbacks: Changing screen locations, screen scalings, non-maximized applications etc. lets the test break as the same screen location doesn't point to the same UI element anymore. Additionally, every time the layout of the SUT changes, the test also breaks.

The next-best version is to test with an image-based approach: Clicking on a screen location that *looks like* a specific source (like a screenshot) leads to much better results. However, some of the limitations still remain: chainging, screen resolutions, changing layout etc. Additionally, some use cases that require more sophisticated interactions with strings can only be achieved using OCR (Optical Character Recognition).

A typical appraoch to circumvent all those limitations is to use object based

automation. An action in the test will be performed on a specific user interface element, where the challenge is how to find this element on the screen. A common approach is to do so by using accessibility technologies. In [15], the authors explain this approach in more detail: Many applications expose internal user interface information to external tools, mainly to provide access for disabled users to this application. Screen readers normally use this accessibility information.

This approach has developed to be the main approach in industry today as it mitigates most of the issues of the other methods. Ranorex, too, uses GUI object recognition based on accessibility information.

Graphical user interfaces are typically organized in a tree-like structure based on their position in relation to each other. If an element lies *within* another element, then it is represented as a child element of this parent node. A tab page might be a child of a tab page list, and a tab page might contain children like text elements, buttons or containers that itself might again contain buttons, checkboxes etc.

There is the XPath notation[22] that is typically used to select nodes or nodesets in an XML document. Since XML-documents have a similar tree-like structure, this XPath notation is well-suited to also select elements in a GUI. Ranorex has built its own version of the XPath with very similar syntax and semantics, the *RanoreXPath* that will be explained in section 4.2.4 on page 30.

Most of the time, application developters will not develop the UI controls themselves, but instead rely on GUI frameworks. Those frameworks have accessibility implemented already in most cases, therefore developers don't have to worry about accessibility themselves normally. Famous GUI frameworks include WinForms[23], WPF[24], Qt[25], Java Swing[26], Delphi[27], Xamarin[28],

[22] Explanation of the XPath from the W3C web page: `https://www.w3schools.com/xml/xpath_syntax.asp`

[23] WinForms: `https://docs.microsoft.com/en-us/dotnet/framework/winforms/`

[24] WPF: `https://msdn.microsoft.com/en-us/library/aa663364.aspx`

[25] Qt: `https://www.qt.io`

[26] Java Swing: `https://docs.oracle.com/javase/tutorial/uiswing/index.html`

[27] Delphi: `https://www.embarcadero.com/products/delphi`

[28] Xamarin: `https://docs.microsoft.com/en-us/xamarin/`

HTML5[29], and many others.

Web Accessibility

The web is inherently built with accessibility already in mind. HTML itself is a markup language to describe web content, which is exactly what accessibility is about. It separates content from layout. Browsers are implemented to interpret this semantic meaning behind HTML documents. They, for example, know that something that is tagged as button should be styled like a button, should be focusable and clickable. This behavior is already very near to what object based test automation is about. Still, the better the HTML is written with accessibility in mind, the easier its automation is[30].

The big difference to non-web technologies is that the HTML document itself is visible, while the internal state of a desktop application isn't visible from the outside and has to be read with other means, for example doing code injection. This means that automating web pages does not need as complex automation tools as many desktop applications.

For actually automating web pages, the Selenium[31] framework has developed to be the predominant framework. It is open-source and supported by all major browsers.

HTML is the base for many mobile applications, too, so automating mobile web application works similar to regular web applications as they build upon the same technology. On mobile platforms, Appium[32] has developed to be the main open-source automation technology.

[29]HTML5: `https://www.w3schools.com/html/html5_intro.asp`

[30]More information on well-implemented accessibility features, as suggested by the W3C, can be found here: `https://www.w3schools.com/html/html_accessibility.asp`

[31]Web page of the Selenium project: `https://www.seleniumhq.org`

[32]Web page of the Appium project: `http://appium.io`

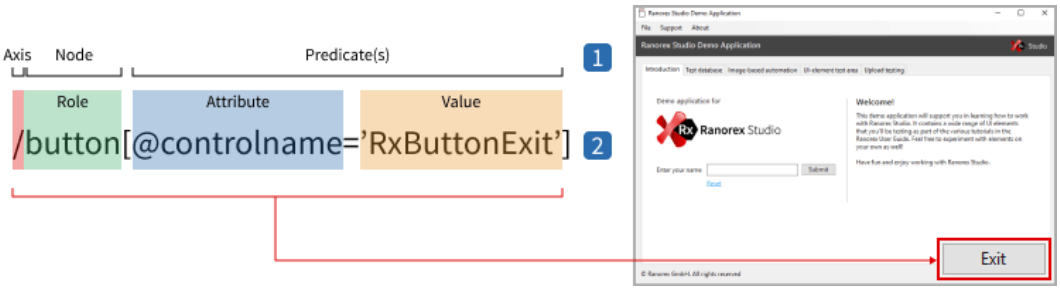


Figure 4.7: This image shows a RanoreXPath and the corresponding UI element that it identifies in a small sample application. It also shows the different syntactic parts of the RanoreXPath. Image taken from the Ranorex user guide with friendly permission of the Ranorex GmbH.

4.2.4 The RanoreXPath

The introduction into the RanoreXPath notation from the Ranorex user guide goes like this:

> A RanoreXPath expression is primarily used to uniquely identify UI-elements within a desktop, web, or mobile application. The syntax draws on the XML description syntax W3C XPath. In other words, RanoreXPath is a subset of XPath with necessary selected extensions.
>
> Every UI-element can be described by a unique RanoreXPath description. RanoreXPath is used to describe, search, identify and find UI-elements within an application. The tools for creating and editing RanoreXPath are the path editor and Ranorex Spy, presented within separate chapters.[33]

As explained in section 4.2.3 on page 26, a GUI can be seen as a tree structure of UI elements, similar to an XML tree. Therefore, an XPath-based notation seems reasonable. A RanoreXPath can be constructed by simply chaining links of chain elements together. A chain link itself consists of three parts: Axis, Node and Predicate(s).

[33]Ranorex User Guide: `https://www.ranorex.com/help/latest/ranorex-studio-advanced/ranorexpath/introduction/`

Axes Axes (singular: Axis) specify the navigation direction within the UI tree. A slash (/), for example, means that this chain link is a direct descendant (child) of the parent. A double slash on the other hand means that the link is a descendant on any layer below the ancestor, etc. There are other axes, too, that can map more complex relationships, for example `::following-sibling` that matches all items on the same layer after the link before, or the double dot (`..`) that *goes one level up*, similar to standard Unix notation when navigating directories.

Role (Node) The node (in Ranorex typically called *role*, directly mapping to the internal *adapter-types*) specifies the type of element to identify in this link. A role can be anything that a GUI element represents (like `button` or `text`), or a special identifier like the asterisk (`*`) that can match any role.

Predicate The predicate is optional and is nested between the brackets `[` and `]`. The predicate can be a simple number that just is an index in the set of all elements that were found with the path before, but usually it consists of an attribute/value pair. UI elements often have many attributes with specific values that can all be used to uniquely identify it and distinguish it from all other UI elements.

Wildcard operators There are several wildcard operators that make parts of the path to the leaf element in the tree optional. For example, `/*` means *any* element exactly on tree level further down can be matched, and the already-explained `//` matches any descendant of the parent. Usage of the wildcard operators enables the tester to make shorter and more robust paths, but they also impact performance as Ranorex has to search in more branches for the correct item. More information about this trade-off between robustness and performance can be found in section 7.2.3 on page 56.

4.3 IronPython

IronPython is the framework that makes the RanorexLibrary possible in the first place. It provides the middle layer between Ranorex and the Robot Framework. A smooth interaction between these two systems seems almost impossible: .NET is a framework developed and maintained by Microsoft, aimed at the C# programming language. Its core is the CLR (Common Language Runtime) that offers things like a just-in-time compiler, memory management and security features. This world is strongly influenced by the Windows operating system and streamlined to be used for compiled desktop applications.

On the other hand, there is Python, an open source, dynamically and strongly typed programming language that is interpreted instead of compiled. It is OS-independent and is completely incompatible with the .NET world. Since the Robot Framework itself is a Python script, and the Ranorex API is a set of .NET library files, the only possible way to make them work would be to use complex language bindings and then inter-process-communication. Luckily, there is IronPython. IronPython solves (almost) all the issues that come with having these completely diverse frameworks and programming paradigms. It enables these two worlds to work together.

This introduction to IronPython is mainly based on [1], the standard introduction book into IronPython.

4.3.1 What is IronPython?

> IronPython is an open source implementation of Python for .NET. It has been developed by Microsoft as part of making the CLR a better platform for dynamic languages. In the process, they've created a fantastic language and programming environment.[34]

IronPython consists of the IronPython engine and some helper tools. The engine compiles Python code into Intermediate Language (IL) that runs

[34] From [1]

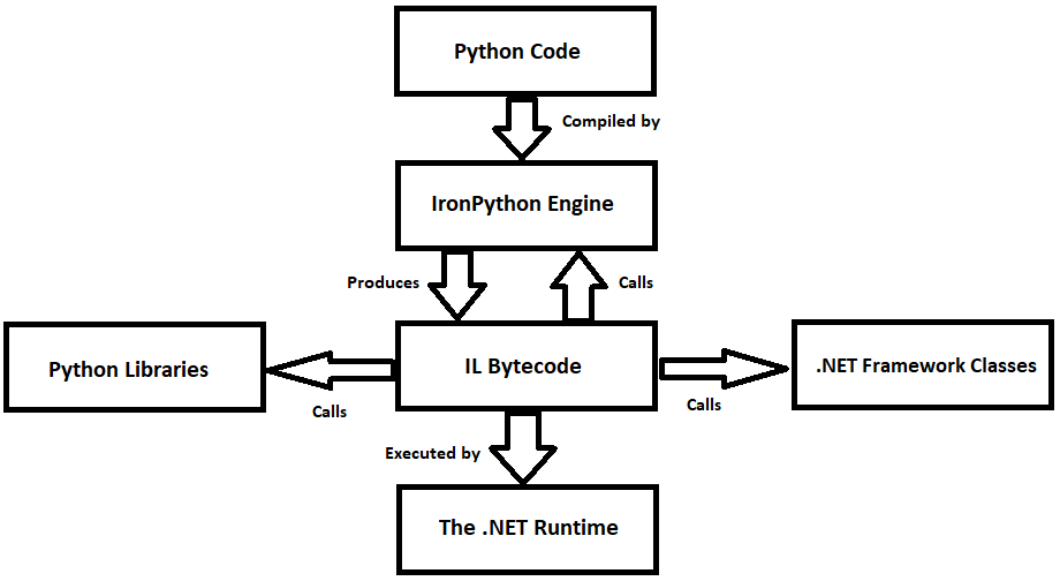


Figure 4.8: This figure shows the IronPython infrastructure. IronPython fills the slot between Python (language) code and the .NET framework by making it possible to call both Python labraries and .NET library files.

on the CLR. Additionally, IronPython can be compiled (!) to make binary-only files. Figure 4.8 shows how the IronPython engine slots into the space between Python and the .NET framework.

The generated IronPython assemblies are still compiled .NET code and therefore .NET classes can be accessed without any need for explicit type conversions. Python in IronPython feels very similar to C# code—and the other way around.

Listing 4.1 on page 34 shows a very simple example of .NET code running in an IronPython environment. First, the `clr` module is imported that handles all the interaction with .NET assemblies. Using `clr`, assemblies can be added with functions like `AddReference()` or `AddReferenceToFileAndPath()`. And the real magic is in the next lines: Classes from the .NET assemblies can now be imported like any Python module using the `import` functionality. In this example, WinForms classes are imported. A small form is created and run. The output of this small script is a WinForms application that starts on the Windows desktop. Classes in .NET assemblies behave just like Python

```
>>> import clr
>>> clr.AddReference('System.Windows.Forms')
>>> from System.Windows.Forms import Application
>>> from System.Windows.Forms import Form
>>> form = Form()
>>> form.Text = 'Hello World'
>>> Application.Run(form)
```

Listing 4.1: This small example (directly taken from [1]) shows how .NET assemblies can be accessed using the IronPython interpreter. The huge differences in technology, paradigm and underlying technology are completely transparent.

classes, and can be used like Python classes.

There are some language-specifics that are not so easy to translate to the other language. For example, the `out` keyword only exists in C#, but not in Python. A C# function can have an `out` parameter which would indicate that this parameter should be changed in the function, making it similar to a *call by reference*. Python doesn't know this concept, and passing an immutable object (like a string) to this function couldn't work. IronPython does handle this fact in the background, though, by converting the `out` parameter into an additional return value—something that C# doesn't know how to do.

4.4 Example Integration: SeleniumLibrary

To understand how the RanorexLibrary has to look in the end, it can be beneficial to look into other integrations into the Robot Framework that already exist and proved to work really well. Maybe the most popular external library for Robot is the SeleniumLibrary.

Selenium is a test automation framework and standard for web tests. To make it work, a specific driver has to be installed next to the browser that runs the automated test.

> SeleniumLibrary is a web testing library for Robot Framework that utilizes the Selenium tool internally. The project is hosted on GitHub and downloads can be found from PyPI.

> SeleniumLibrary works with Selenium 3. It supports Python 2.7 as well as Python 3.4 or newer. In addition to the normal Python interpreter, it works also with PyPy and Jython. Unfortunately Selenium is not currently supported by IronPython and thus this library does not work with IronPython either.
>
> SeleniumLibrary is based on the old SeleniumLibrary that was forked to Selenium2Library and then later renamed back to SeleniumLibrary. See the Versions and History sections below for more information about different versions and the overall project history.[35]

The SeleniumLibrary is a *dynamic* Robot library that can be installed using `pip`. Typical shortcomings of this library are that everything that exists outside of the document object model (DOM) can't be automated directly, but on the other hand it provides a huge amount of web specific keywords.

The most interesting aspect of this integration as a benchmark for the RanorexLibrary is which keywords it implements. A full keyword documentation is available on the internet[36].

Locators A very interesting question in this comparison is how the SeleniumLibrary addresses elements in the UI. This is already the biggest difference between the RanorexLibrary and the SeleniumLibrary: The SeleniumLibrary supports finding elements based on different strategies like id, XPath or CSS selectors. How elements should be found can be either stated explicitely by the tester or is determined implicitely. In contrast, Ranorex does only support one way of finding elements: the RanoreXPath[37].

[35] From `https://github.com/robotframework/SeleniumLibrary/blob/master/README.rst`

[36] SeleniumLibrary Documentation: `http://robotframework.org/SeleniumLibrary/SeleniumLibrary.html`

[37] However, Ranorex does support a shortened syntax for directly finding web elements by id, which could technically count as another way of addressing elements, although this happens still within the RanoreXPath syntax.

Cookies The SeleniumLibrary provides keywords (like `Add Cookie`) that specifically deal with managing cookies. This is very web specific, similar functionality is not implemented in Ranorex, apart from a "clear cookies" option in the `Open Browser` action.

Clicking Elements In the RanorexLibrary, there is only one keyword for a normal click action, and it does not matter which type the clicked element has. On the other hand, the SeleniumLibrary offeres different keywords: `Click Image`, `Click Button`, `Click Element`, etc. This is one of the main benefits of Ranorex: It abstracts the underlying technology completely.

WebDriver and Browser There are several keywords that are specific for testing using Selenium (like creating WebDriver instances) and that deal with manipulating browser windows in finer detail, like `Close Browser`, `Close All Browsers`, etc.

Validations The SeleniumLibrary builds validations directly in the library, for example with keywords like `Element Attribute Value Should Be` or `Element Should Be Disabled`. Since validation actions in Ranorex are tied to a repository item and the repository is not implemented within the RanorexLibrary, validations could only implemented manually directly within the RanorexLibrary. However, since Robot offers keywords like `Should Be Equal` in its `BuiltIn` library, these validations can be easily done by the tester themselves by retrieving information from the system under test manually.

Layout-specific keywords Ranorex tries to ignore an application's layout, while the SeleniumLibrary offers easy ways to access it with keywords like `Get Horizontal Position` or `Get Element Size`. All of these contradict the defining philosophy of Ranorex of being a functional test automation tool. Thus, similar keywords are not implemented in the RanorexLibrary[38].

[38]These keywords are even strange in Selenium, as Selenium also works object-based. There are much better tools for checking an application's layout or appearance than

Higher order keywords The SeleniumLibrary offers many keywords that could be considered to be *higher level*, like `List Selection Should Be` or `Radio Button Should Not Be Selected`. Those are easy to create if needed using other, more basic keywords. In the RanorexLibrary, these keywords would not make much sense since Ranorex knows hundreds of different control types (like Radio Buttons). Providing special keywords for all of them would bring the number of implemented keywords into the thousands. Instead, Ranorex has a lower-level approach: being able to access any attribute from any type of control in any UI technology.

Conclusion The SeleniumLibrary offers many more keywords than the RanorexLibrary, however, many of them are not even necessary in Ranorex (`Textfield Value Should Be`, `Title Should Be`, `Table Header Should Be` etc. would all just use the same keyword in the RanorexLibrary), and some are only geared towards web testing. The RanorexLibrary works on a lower abstraction layer, and while Selenium aims to mimic *user actions*, Ranorex simulates mouse and keyword actions. Thus, the two libraries look a bit differently in the end, although they do similar things.

Selenium or Ranorex.

5 Requirements for the Integration

This chapter presents the requirements that are in place for the RanorexLibrary. It is not meant as a full requirements document, but instead as a collection of must-haves and nice-to-haves. The implementation of the RanorexLibrary followed the requirements listed in this chapter.

5.1 Licensing

The RanorexLibrary slots into an environment consisting of several different technologies and libraries. The RanorexLibrary is written with all the restrictions in mind that come with these licenses. End users should also be aware that all these technologies are published under a license.

Ranorex Ranorex is the only tool with a proprietary license. To be able to use the RanorexLibrary, a full Ranorex installation and valid Ranorex licenses are necessary. More information on Ranorex licensing is provided in chapter 4.2.1 on page 22.

Robot Framework The Robot Framework[1] is distributed under the Apache License Version 2.0[2]. The Robot Framework itself is only used and referenced, but but its source is never changed when using the RanorexLibrary.

[1] License information on GitHub: `https://github.com/robotframework/robotframework/blob/master/LICENSE.txt`

[2] Full license text: `http://www.apache.org/licenses/LICENSE-2.0`

The end user has to make sure to use the Robot Framework within the (albeit very few) license restrictions.

IronPython IronPython is also distributed under the Apache License Version 2.0[3]. IronPython includes the `zlib.net` library that uses another license that allows to use this library within IronPython. IronPython is also just used as a Python environment and therefore the end user has to make sure to stay within the license restrictions.

RanorexLibrary The RanorexLibrary itself is licensed using the MIT license[4].

5.2 The Ranorex Repository

One of the first questions to answer when making an integration between Ranorex and another test automation tool is about the layer to base the integration on: pure API (layer 1 in Ranorex) or repository (layer 2 in Ranorex)[5].

The repository would provide many advantages by providing another layer of abstraction. Having the repository, the actual item a keyword would work on would be the repository item instead of a pure RanoreXPath. Having the abstraction of a UI element in its own separate place would make maintenance of the test project much easier as it would provide a single point of maintenance for all UI changes. If it is not possible to use the Ranorex repository, the end user would have to use best practices (like the Page Object Pattern) themselves in order to ensure a maintainable test design.

[3]IronPython Licensing information: `https://ironpython-test.readthedocs.io/en/latest/license.html`

[4]License text of the RanorexLibrary: `https://github.com/Thomas-Gruber-90/RanorexLibrary/blob/master/LICENSE`

[5]More on the layer structure of Ranorex is in chapter 4.2.2 on page 23.

5.2.1 General Workflow

A general worklow when using the Ranorex repository looks like this:

1. Create a repository using the Ranorex Spy.
2. Export the repository as .cs file.
3. Compile the file manually into a .dll library.
4. Tell the RanorexLibrary where to find the repository files.
5. Use repository item names as parameters for keywords instead of RanoreXPaths.

This means that large parts of this process wouldn't be handled by the RanorexLibrary, but by the user. Creating a Ranorex repository using the Spy is a rather simple process. However, Ranorex does only provide the exporting functionality into a `.cs` file, not directly into a `.dll` library format. This means that the end users have to know how to compile this file themselves, most likely using the command line. When using CSC[6], the correct batch file could look like this:

```
csc /t:library MainPageObject.Repository.cs
    /r:"\path\to\Ranorex.Core.dll"
    /platform:x86
```

This would also require to specifically telling the compiler where the `Ranorex.Core.dll` file is located and for which platform to build. And every time that repository changes, this process has to be done again. Then, the file has to be saved to a folder and this folder path would have to be given as a parameter to the RanorexLibrary.

5.2.2 Rationale for not Implementing the Repository Functionality

The process of creating a Ranorex repository, then exporting it and compiling it manually, is already relatively cumbersome for the end user. In addition, from the implementation standpoint of the RanorexLibrary, there

[6]Command line build tool for .NET: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe`

are additional problems: The namespace of the repository plus the repository name itself are both just known at runtime, but the symbols have to be imported in order to use them. This would, in Python, mean to parse all files in the repository directory for those names, and then importing them at runtime dynamically, making them global for all modules. The alternative would be to force the end user to explicitely give all the namespace and repository names explicitely. This, again, would make the import of the RanorexLibrary into a Robot test suite cumbersome as it would not require just one parameter (the location of the Ranorex library files), but potentially many.

I have built a small proof of concept that uses the Ranorex repository files, however, due to all the shortcomings and problems with the solution, I decided to not include support for the Ranorex repository.

5.3 Desktop Testing

On the one hand, the big strength of Ranorex is desktop testing. Ranorex supports many of the most common desktop user interface technologies seamlessly. A mouse click on an element has the same look and feel in Ranorex, no matter if it is performed on a Java Swing panel or a Delphi table cell. On the other hand, there are very good solutions specifically for web testing (Selenium) and mobile testing (Appium). However, none of the available Robot libraries support more than a single desktop technology at a time. There are libraries for (for example) Java Swing, but nothing that is suitable for real end-to-end testing.

As this is the strength of Ranorex, the main goal of the RanorexLibrary is to also focus strongly on desktop testing. Luckily, desktop testing has fewer of the typical pitfalls than mobile or web testing.

The main goal of the RanorexLibrary was to transport the low-level functionality of Ranorex to Robot directly and without cutting too much. This means that all the actions that Ranorex Studio provides should be included and implemented, and with all possible parameters.

There are a few exceptions, however: Robot has its own logging, so the `Log` action doesn't make any sense. If a user wants to use Robot, they also want to use the Robot reporting instead of the Ranorex reporting capabilities. The `Delay` action has a similar problem: Robot provides the `Sleep` keyword in its `BuiltIn` library that has the same functionality, so there is no point in implementing this within the RanorexLibrary.

There are a few keywords that are not yet implemented, that might make sense and could be added later:

- *Invoke Action*: Gives flexibility and deep control over UI components, but is more complex in its usage.
- *Screenshot*: This Ranorex action always takes a screenshot and puts it into the Ranorex report. This doesn't make sense within Robot, so another screenshot keyword, that can't use the corresponding Ranorex version, could be necessary.
- *Create Snapshot*: A snapshot of a user interface is mostly used to communicate with the Ranorex support team about issues. It can be created within Ranorex Spy directly, so this keyword would be mostly unnecessary.
- *Mouse Wheel*: Most of the time there are superior ways of navigating a page as using the mouse wheel is not very robust and contradicts the main design principles of object based automation.

5.4 Web Testing - Advantages over the SeleniumLibrary

If a user only wants to do web testing, there already is the SeleniumLibrary that builds upon Selenium[7], the de-facto standard for web testing. Under the name *WebDriver* it has also an official recommendation in the W3C[8].

However, there are a few challenges that can not be solved with Selenium (easily) that pose no problem for Ranorex. Not every functionality that

[7]Homepage of the Selenium project: `https://www.seleniumhq.org`
[8]WebDriver Recommendation: `https://www.w3.org/TR/webdriver1/`

is important in web testing is a part of the web application itself. For example, downloading and saving a file from a web page usually opens a file-save dialog—that is browser-specific. Without using external libraries, it is not possible to automate this dialog with Selenium, and therefore the SeleniumLibrary also does not offer this capability. However, since Ranorex can not only automate the web page itself, but also the browser (as it is just another desktop application), this scenario is no challenge for Ranorex.

Thus, in some end-to-end testing scenarios, there are things that are possible with Ranorex where Selenium struggles. This also makes it necessary to include web testing capabilities in the RanorexLibrary, especially since web applications become increasingly important, even if they are disguised as desktop applications (for example in CEF[9]).

5.5 Mobile Testing

Testing mobile devices is significantly more complex than desktop or web testing. Many different operating system versions, vastly different hardware, a fast-paced update cycle, the mixture of native and web apps, a necessary connection to a test automation system etc., are all factors why mobile testing proves to be rather complicated.

One of Ranorex's strenghts is that mobile testing feels very similar to desktop or web testing. There is the `Touch` action instead of a `Click` action, but they actually even match to each other, and *clicking* a mobile element instead of *touching* would just work within Ranorex.

Adding a device as automation endpoint to a Robot test is surprisingly easy: The keyword `Add Device` adds the device to the test runner instance, and the keyword `Run Mobile App` starts an app. Automating within this app then works without any changes to the usual behavior, there are just a few specific special keywords (like `Touch`, `Double Tap`, or `Long Touch`).

[9]Project page of the Chromium Embedded Framework: `https://bitbucket.org/chromiumembedded/cef`

6 Implementation Detail

This chapter gives an overview over the actual implementation of the RanorexLibrary. It will touch both the general aspects like the fact that it is a Static Robot Library and how the Ranorex assemblies are loaded, but it will also go into some keywords and explains their internals based on some representative examples. Since `Double Click`, `Right Click`, and `Click` are very similar, only their common parts are described.

6.1 General Details

The RanorexLibrary is implemented as a *static* Robot library as described in the official Robot documentation[1]. This means that all keywords are defined by their corresponding function name in the Python library script. Ignoring underscores, the function name `run_Application()` becomes the Robot keyword `Run Application`. Without switching to a *dynamic* or *hybrid* library, there is no way to change this behavior. Since this is okay and makes the code easier, I have decided to stick with a static library.

The general setup of the library is placed in a separate file. There, all the parts that deal with importing Ranorex into the Robot library are listed. This means that first, `clr` is imported and then all the necessary references are added. First, the `System.Windows.Forms` library is needed, and then all the needed Ranorex references are added using their absolute path. This path is given by the user when importing the RanorexLibrary.

In the initialization of the RanorexLibrary object, the Ranorex Resolver is initialized and the core is set up. These two lines are necessary since

[1]Definition of a static Robot library: `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#creating-static-keywords`

Ranorex doesn't install into the GAC anymore, but still needs to find all the library files. This is similar to how it is suggested in the official Visual Studio integration guide[2]. Without these lines, Ranorex is likely to not work correctly after version 8.0.

The rest of the library is split between actual keyword implementations and helper functions. The latter are distinguished from the keywords by a leading underscore (_) as these functions are ignored by the Robot engine.

6.2 Keywords

The keywords typically have a simple *core functionality* that is a direct Ranorex API call, but they also consist of parameter management. For example, a parameter that is passed to a keyword function always appears as `string` value. In order to have a variable number of arguments, some of these are optional. However, Robot allows to have "empty" keyword argument cells. Those are passed to the RanorexLibrary as empty strings, so the RanorexLibrary has to reassign the default value to an empty string parameter.

This section will in most cases only describe the actual core of the keyword function, not the surrounding parameter management. Additionally, all keywords implement a small logging function that is also omitted here.

Runing and Closing Applications Starting an application within Ranorex is usually done using the `Host.Local.RunApplication` function of the Ranorex API and is implemented in the `Run Application` keyword.

```
Ranorex.Host.Local.RunApplication(appname, arguments,
    workingDirectory, maxim)
```

Although it is possible to start an application using a double click on the icon, and many testers do this, is is not a best practice, because this would make the test reliant on the Windows desktop layout that the tester

[2]Example code from the user guide: `https://www.ranorex.com/help/latest/interfaces-connectivity/visual-studio-integration/`

typically has no control over. Additionally, a `Double Click`'s purpose is not as obvious as an explicit `Run Application` keyword.

Closing an application is similar, however, as parameter the user doesn't give the path to the executable anymore, but the RanoreXPath of any UI control within the application they want to close.

```
return Ranorex.Host.Current.CloseApplication(ranorexpath,
    intGracePeriod)
```

Closing a browser and a mobile app works the same as closing a desktop application, but starting them looks differently. The keyword `Start Browser` has another API function, because it takes different parameters (like if the cache has to be cleared or if the browser should be started in incognito mode):

```
Ranorex.Host.Current.OpenBrowser(url, browser, browserArgs,
    strtobool(killExisting), strtobool(maximized), strtobool(
    clearCache), strtobool(incognitoMode), strtobool(
    clearCookies))
```

Starting a mobile application is similar, but it also has its own API function and keyword, because the endpoint (the device that the app has to run on) has to be stated explicitely, something that is not necessary for the other two:

```
Ranorex.Host.Local.RunMobileApp(endpoint, appname, True)
```

Click Actions There are several `Click` keywords that do actions like clicking, double clicking, right clicking, mouse down, mouse up, etc. They all default to a `_click()` helper function (plus, if necessary, a `move()` function). The click looks like this:

```
exec("Ranorex.Unknown(ranorexpath).Click(" + mousebutton + ",
    " + location + ", " + "int(" + count + "), " + duration +
    ")")
```

The move action is sometimes performed before another action and looks like this:

```
exec("Ranorex.Unknown(ranorexpath).MoveTo(" + location + ", "
    + duration + ")")
```

Both these keywords have in common that they are called using the `exec()` function in Python. This makes it possible to interpret some symbols as Python code that are passed as string parameters. The other interesting thing about it is the `Unkown` adapter. This adapter is Ranorex' way of saying "I don't care what it is, but perform this action on it." Ranorex uses this adapter internally for performance reasons, and it also makes the code rather simple. Without this option, it would be necessary to either create an Ranorex UI element or to implement the function for all adapter types separately.

Touch actions for mobile devices (like `Touch`, `Double Tap`, etc.) are very similar to clicks, and internally within Ranorex even default to clicks.

Key actions Key actions in Ranorex are either `Key Shortcuts` or `Key Sequences`. The main difference is that `Key Shortcuts` are meant to be executed without a specific UI element as target, therefore the code looks like this:

```
Ranorex.Keyboard.Press(sequence)
```

This has the disadvantage that Ranorex does not know if it has to wait for something to happen, for example a UI to be fully loaded or a UI element to be active. Therefore, when using this action, it is often necessary to use an explicit waiting action to make it work consistently. This keyword is mainly meant for navigating through the application with function keys, copying and pasting, etc.

The `Key Sequence` is similar, but it requires a RanoreXPath and therefore a UI element to put that key sequence into. It is mostly used to type strings into text fields, and looks like this:

```
Ranorex.Unknown(ranorexpath).PressKeys(value)
```

This keyword is safer to use as it knows how to wait for the used UI element to be ready for text insertion.

Getting and Setting Values Sometimes it is necessary to get values from the UI and to also set them. The former is often used for conditional execution and for validations. The latter enables changing of UI attributes, even if they are not changable normally. Since the Getter function is overloaded in the Ranorex API, the RanorexLibrary has to explicitely tell it what version to call. The Setter function only works on an Element object, therefore this Element has to be used for setting a value.

```
return Ranorex.Unknown(ranorexpath).GetAttributeValue[str](
    attribute)

Ranorex.Unknown(ranorexpath).Element.SetAttributeValue(
    attribute, value)
```

Waiting for an element Waiting for an element to exist is an important part of GUI testing. However, Ranorex doesn't offer the waiting functionality in a nice way in its API, so this actually has to be implemented manually:

```
intRanorexpath = Ranorex.Core.RxPath(ranorexpath)
intDuration = Ranorex.Duration(int(duration))
newElement = None

elementFound, newElement = Ranorex.Host.Local.TryFindSingle(
    intRanorexpath, intDuration)
if not elementFound:
    raise AssertionError('Element hasn\'t been found within
        the specified timeout of ' + duration + 'ms: ' +
        ranorexpath)
```

Waiting for other things (like a state or an element becoming visible) is more complex and therefore not yet implemented in this version of the RanorexLibrary.

Here, the Ranorex API does use the `out` keyword of the C# programming language. Python does not have a similar language feature, but IronPython maps this `out` parameter to a second return value. This feature really shows the power of IronPython.

Validations Validations belong to the most important part of testing. Without asserting a system state, a test would not be a test but a simple flow control. Unfortunately, a validation within Ranorex requires to have a repository item. This means that the internal validation functionality of Ranorex can't be used at all. The keywords `Validate Attribute Equal` and `Validate Attribute Not Equal` are implemented, but most other validations can be achieved easily on the Robot keyword level as there are several keywords for validations. The tester has to get the information they need explicitely, however, for example using the `Get Attribute Value` keyword.

The implemented version looks like this:

```
varToVal = Ranorex.Unknown(ranorexpath).GetAttributeValue[str
    ](attribute)
if not varToVal == value:
    raise AssertionError("Elements are not equal. Expected " +
        value + ", but got " + varToVal + " instead.")
```

Mobile Keywords When testing mobile applications, most features work similarly to in desktop applications. Touches are like clicks, key sequences stay key sequences. Starting mobile apps has its own keyword, and closing works like in web or desktop applications.

However, before being able to run a test on a mobile device, it is necessary to *connect* that device to the Ranorex test. This is done using the `Add Device` keyword[3].

```
platform = "Ranorex.Core.Remoting.RemotePlatform." + platform
typeName = "Ranorex.Core.Remoting.RemoteConnectionType." +
    typeName
exec("Ranorex.Core.Remoting.RemoteServiceLocator.Service.
    AddDevice(\"" + name + "\", " + platform + ", " + typeName
    + ", \"" + address + "\")")
```

[3]Fun fact: Finding and implementing this simple code snippet cost me a lot of sweat and tears.

7 Setup and Tutorial

This chapter explains how to set up the whole infrastructure that is needed to run Robot tests that use the Ranorex API for automating graphical user interfaces. There are some prerequisites that have to be fulfilled for everything to run and several tools have to be installed and configured correctly.

This chapter also gives a small example plus some tutorial on how to set up tests, best practices and where to find documentation and help.

7.1 Setting Everything up

Ranorex The first requirement is to have all the Ranorex requirements in place as listed here on the official Ranorex web page[1]. Then you also have to have access to the required Ranorex .dll files in your file system. The required files are:

- Bootstrapper.dll
- Contracts.dll
- Controls.dll
- Core.dll
- Core.Resolver.dll
- Core.Injection.dll
- Core.WinAPI.dll
- Plugin.Cef.dll
- Plugin.CefHost.dll

[1]Ranorex' system requirements: `https://www.ranorex.com/help/latest/ranorex-studio-system-details/system-requirements/`

- Plugin.ChromeWeb.dll
- Plugin.FirefoxWeb.dll
- Plugin.Flex.dll
- Plugin.Java.dll
- Plugin.Mobile.dll
- Plugin.Msaa.dll
- Plugin.Office.dll
- Plugin.Qt.dll
- Plugin.RawText.dll
- Plugin.Sap.dll
- Plugin.Uia.dll
- Plugin.Web.dll
- Plugin.WebDriver.dll
- Plugin.Win32.dll
- Plugin.Winforms.dll
- Plugin.WinformsProxy.dll
- Plugin.Wpf.dll
- Plugin.WpfProxy.dll

To actually run a Ranorex test, a valid Ranorex license is also required. For more information, see section 4.2.1 on page 22.

IronPython The Robot Framework can run on each of plain Python, Jython or IronPython. Since Ranorex is a .NET API and its library files get imported into the RanorexLibrary, it is necessary to have an IronPython installation. First, IronPython has to be downloaded[2] and installed. At the time of writing, the most recent version of IronPython is 2.7.8. As the `elementtree` module that comes with IronPython is broken[3] in that version, the `elementtree preview 1.2.7 release` also has to be installed[4]. The `elementtree` module can be installed by downloading the source, unzipping it and running `ipy setup.py install` on the command prompt in the created directory.

[2]Download from the official web page: `http://ironpython.net`
[3]See `https://github.com/IronLanguages/main/issues/968` for more information.
[4]It can be downloaded from here: `http://effbot.org/downloads/#elementtree`

PATH Although not strictly necessary, setting the Path system environment variable is recommended to make interacting with Robot easier. The path to the IronPython installation itself has to be added. It is by default located at `C:\ProgramFiles\IronPython2.7`. As the Robot scripts are best installed into `C:\ProgramFiles\IronPython2.7\Scripts`, it is also best to add this directory to the Path.

Robot Framework The Robot Framework can be installed using `pip`. First, `pip` has to be activated:

```
ipy -X:Frames -m ensurepip
```

Then, the framework can be installed using `pip`:

```
ipy -X:Frames -m pip install robotframework
```

By default, the two scripts `robot.py` and `rebot.py` are installed into the `Scripts` directory in the IronPython installation directory.

Writing scripts Writing Robot scripts requries nothing more than a text editor and access to a command line. Many editors and IDEs offer syntax highlighting and additional functionality for the Robot syntax, like Eclipse, Emacs, Vim and VS Code. As soon as there is a Robot script file in valid syntax, it can be executed using

```
ipy32 -m robot myTest.robot
```

Instead of `ipy`, `ipy32` has to be used, because the Ranorex .dll files themselves are also 32-bit library files. If everything is installed correctly and all paths are set, the test should now start and execute.

7.2 Best Practices

Like all frameworks, the RanorexLibrary can be used effectively and ineffectively. How well a test works does not only depend on the tools used, but mainly on how the tests are created and planned. There are a few best practices that make a test created with the RanorexLibrary better and thus should be used in every test scenario.

7.2.1 Modularization

As in every software project, it is always a good idea to encapsulate parts that occur often into their own reusable modules. A very common example might be opening and logging in to an application: This might occur in every single test case or test suite and might consist of several different low level actions. Instead of putting them into every test case where they are needed, it is generally better to put them into a module and use that module instead. If something changes in this set of actions (opening an application and logging in), then the module has to be only changed once, and not in every occurence.

Robot provides a very natural way of constructing modules by using higher order keywords. A keyword itself can consist of a set of other keywords, and they in turn might also consist of keywords.

For example, the user-generated keyword `Start SUT and Login` might consist of the two keywords `Start SUT` and `Login`. In Robot notation this would be as simple as this:

```
*** Keywords ***
Start SUT and Login
    Start SUT
    Login
```

The two keywords `Start SUT` and `Login` can in turn be also constructed from lower level keywords.

Splitting the test project into smaller modules gives a single point of maintenance for all parts of the test script, but it also makes tests easier to read and understand. A complex set of instructions that fill a form might consist of many `Click` and `Key Sequence` keywords, but abstracting this into a module with the keyword `Add Table Entry` with the appropriate parameters does not only provide this single point of maintenance, but also makes the test readable and easier to understand.

7.2.2 Page Object Pattern

The page object pattern is an important concept that has developed to be a standard for writing good Selenium tests. It is described in more detail in [14].

The core idea of the page object pattern is to separate the test logic from the user interface itself. The idea originates from Selenium, where the page object pattern splits an action into two different parts: The first part is to generate a low level action in pure (normally) Java code, for example `proceedToCheckout()`. This function internally calls the Selenium API functions that are necessary to proceed to the checkout page, typically by clicking on a button, although this action can consist of more steps. The second step then assembles those functions and arranges them to test cases and a test suite. On this layer, the internals of Selenium and all the technical detail are transparent.

This pattern gets its name from the fact that these functions that abstract the functionality are typically bundled together in a *Page Object*, a file that contains all the actions of a *page* in the browser. One such page might be the front page of the web page, another page might be the shopping cart page. However, other page objects are also possible and common, for example a page object for the main menu, one for the main panel, one for the header, one for the footer etc.

Something similar can be achieved using the Robot Framework and the RanorexLibrary. In a first step, a functionality has to be defined in a keyword. For example, in the Windows calculator, clicking the `Equals` button might be defined in a keyword:

```
*** Keywords ***
Click Equals Button
    Click    /winapp[@packagename<'WindowsCalculator']
                 //button[@automationid='equalButton']
```

In the actual tests, only this keyword `Click Equals Button` would be used. This would abstract the internals of how Ranorex works (for example the RanoreXPath itself) away from the end user. The tester would not have to care about how this keyword works internally, they can just use it. This would also enable a team with different responsibilities to split the work between them: A technically more interested team member could define these keywords, optimize RanoreXPath expressions, and provide these abstract keywords to the testers. These testers then could just use this predefined keyword without having to have any knowledge about Ranorex whatsoever.

7.2.3 Robust Identifiers

Ranorex identifies all the objects solely using the RanoreXPath that is explained in more detail in section 4.2.4 on page 30.

Basically, Ranorex creates a unique RanoreXPath itself automatically. However, there are many ways to tweak and optimize this path that can not be done automatically, especially if more complex functionality should be implemented using one path (for example when using variables within a path expression).

A main principle of the RanoreXPath is that a very general path is very robust, but slow, and a detailed path is fast, but brittle. For example, the path `/dom[@domain='distrowatch.org']//a[@innertext='Manjaro']` made Ranorex look for the element for almost 6 seconds before finding it in a small test, while the path `/dom[@domain='distrowatch.org']/body/table[2]/tbody/tr/td[3]/table[2]/tbody/tr[4]/td[1]/a[@innertext='Manjaro']` made Ranorex finish in about 300 ms on the same machine. While the fist path is significantly slower, it is also more robust against changes in the

application under test. The second path breaks as soon as the order of any of the elements in between changes.

Thus, finding a good balance between maintainability and performance is one of the main tasks of a test automation engineer working with Ranorex. Ranorex can be configured to favor one over the other automatically, but what is best for a specific application always depends on the actual detailed requirements and might differ from path to path.

7.2.4 Data-driven Testing

Most of the time, the test result is not only dependent on the test flow, but also on input data. This data (for example a user name and a password for a login action) can be given in a hard-coded way. However, this contradicts many maintainability ideas. It is better to provide this data from an external data source to provide a single point of maintenance and to be able to run the same test case with different data values without having to change the test case itself.

Ranorex itself would provide data connectors, however, those work on the test suite layer and therefore have no place to slot into the RanorexLibrary. Instead, Robot itself provides basic support for data-driven testing. Most values can be passed to keywords not only in a hard-coded way, but also as variables. These variables can be set in different ways.[5]

7.3 A Small Example

In this section, I will present a very small and basic example of how a Robot test using the RanorexLibrary could look like[6]. The application under test is

[5]More information on usage of variables in Robot keywords can be found in the official documentation: `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variable-files`.

[6]An interesting and—at least for me—surprising fact: I've written this example test in one go without testing it somewhere in between. At the time I tried to run the test for the first time (after it was finished already) I was prepared to face an error that I had to debug,

the official Ranorex test application as this can be downloaded easily[7] and provides an interesting set of UI controls.

Test Scenario In this test case, we will try to validate that, after adding an entry to a small data set, the number of entries correctly counts up to 1. Figure 7.1 on page 59 shows the sample application and the control to validate. Therefore test consists of the following steps:

- Open the application under test.
- Validate that the application is opened.
- Go to the "Test Database" table.
- Validate that the number of entries says `0`.
- Fill the form with data and add an entry.
- Validate that the number of entries now says `1`.
- Delete the entry again.
- Close the Application.

This test is a small, self-contained test that does not require any other tests as prerequisites and that can happen anywhere in the test suite, which already is a nice example of modularization: This test in itself is a self-contained module.

First, we create a file and name it `example.robot` and add the settings section:

```
*** Settings ***
Documentation     This is a sample test case.
Library     RanorexLibrary     C:\\Program Files (x86)\\Ranorex
    8.3\\Bin
```

The `Documentation` part is optional. The only really interesting part is that we add the RanorexLibrary here. We have to tell the RanorexLibrary where to find the Ranorex library files in our file system, and we have to escape the backslashes, because Robot would otherwise interpret them incorrectly.

---

like in every software that I would write normally. But the test worked correctly in the first go already, without any syntax errors or any other complications. This just showed me again how easy Robot is to use.

[7] Link to the test executable file: `https://www.ranorex.com/rx-media/rx-user-guide/latest/download/RxDemoApp.zip`

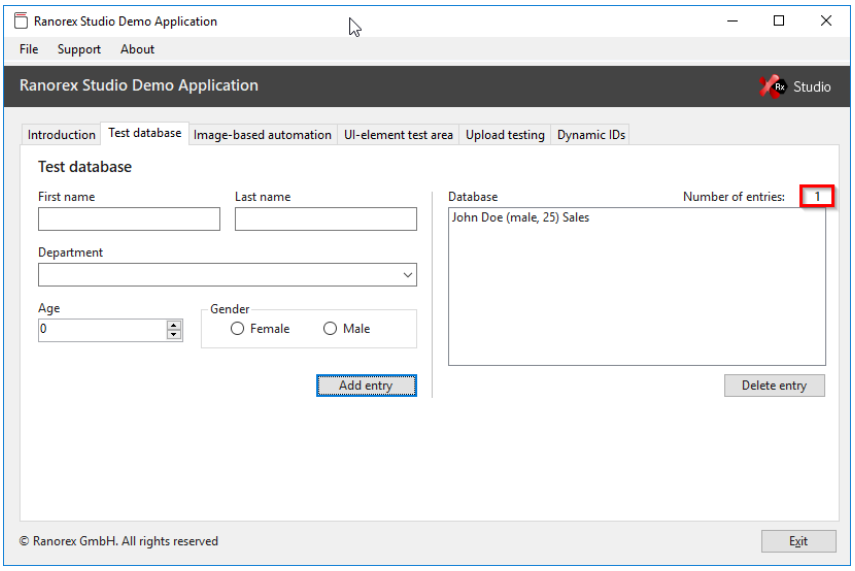


Figure 7.1: This image shows a screenshot of the Ranorex test application. Marked with a red box is the part of the application that should be validated in the sample test case.

Next, although not strictly necessary, I added variables that should tell our tests what the expected values for the validation steps should be. The example might be a bit too simple for this to make much sense, but on the other hand it separates the test logic from the test data more.

```
*** Variables ***
${beforeEntry}    0
${afterEntry}     1
```

Since we already have the test structure, we can just copy it to a new test case more or less verbatim.

```
*** Test Cases ***
Test Number Of Entries Counter
    [Setup]    Run Application    C:\\path\\to\\test.exe
    Validate Open SUT
    Go To Test Database Tab
    Validate Number of Entries    ${beforeEntry}
    Add Entry
    Validate Number of Entries    ${afterEntry}
    Delete Entry
```

```
    [Teardown]    Close SUT
```

Most of these keywords are not even defined yet, so if we would run the test now, Robot would complain and throw an error telling us that a keyword definition was missing. In the next step, we have to define the missing keywords.

Therefore, we create a new `Keywords` section. The first keyword to add is `Validate Open SUT`. This would not be strictly needed as most other actions implicitely wait for the SUT to be opened, but let's add this keyword nonetheless. We have several options to validate an open application, I opted for *waiting for* the `Welcome!` text label to exist.

```
*** Keywords ***
Validate Open SUT
    Wait For    /form[@controlname='RxMainFrame']//text[
        @controlname='lblWelcomeMessage']
```

The next keyword, `Go To Test Database Tab`, is very simple as it only consists of a single click. However, it still makes sense to abstract the actual click and the RanoreXPath away from the actual test since *Go To Test Database Tab* is much easier to understand for a human reader than a *Click* on some element.

```
Go To Test Database Tab
    Click    /form[@controlname='RxMainFrame']//tabpage[
        @accessiblename='Test database']
```

The next keyword, `Validate Number Of Entries`, is very interesting, because it appears in the test case twice. Therefore, it should take the *correct* value as a parameter and compare it to the actual value that it has read from the GUI. It receives this value of the `Text` attribute using the `Get Attribute Value` keyword and saving this value to a variable. The `Should Be Equal As Strings` keywords is defined in the `BuiltIn` library of Robot, not in the RanorexLibrary, showing how elegant it is to mix keywords from different libraries in a single test case.

```
Validate Number Of Entries
    [Arguments]    ${correctNumber}
    ${actualNumber}    Get Attribute Value    /form[
        @controlname='RxMainFrame']//text[@controlname='
        lblNumberOfPersonsNumber']    Text
```

```
    Should Be Equal As Strings    ${correctNumber}    ${
        actualNumber}
```

The keyword `Add Entry` is very long as it fills the form with actual data. In this example, this data is given explicitely, but in a real test environment, those values would probably been passed as variables.

```
Add Entry
    Key Sequence    /form[@controlname='RxMainFrame']//text[
        @accessiblename='First name']    John
    Key Sequence    /form[@controlname='RxMainFrame']//text[
        @accessiblename='Last name']    Doe
    Click    /form[@controlname='RxMainFrame']//button[
        @accessiblename='Open']
    Click    /list/listitem[@text='Sales']
    Double Click    /form[@controlname='RxMainFrame']//text[
        @controlname='upDownEdit']
    Key Sequence    /form[@controlname='RxMainFrame']//text[
        @controlname='upDownEdit']    25
    Click    /form[@controlname='RxMainFrame']//radiobutton[
        @controlname='rdbMale']
    Click    /form[@controlname='RxMainFrame']//button[
        @controlname='btnAddPerson']
```

To bring the application into a clean state again, it is necessary to delete the database entry again. This is done using the `Delete Entry` keyword. Here, a bit of caution is required. Ranorex by itself suggests this path for the database entry list item: `/form[@controlname='RxMainFrame']//listitem[@accessiblename='JohnDoe(male,25)Sales']`. However, RanoreXPath will only ever work if the exact same data is given, but the keyword `Delete Entry` could be used in different test cases with different scenarios. Therefore it is necessary to change this RanoreXPath manually to find a path that does actually address what is meant: the first item of this list.

```
Delete Entry
    Click    /form[@controlname='RxMainFrame']//list[
        @controlname='lstPersonList']/list/listitem[1]
    Click    /form[@controlname='RxMainFrame']//button[
        @controlname='btnDeletePerson']
```

And lastly the application has to be closed again in the `Close SUT` keyword. This is done using the `Close Application` keyword:

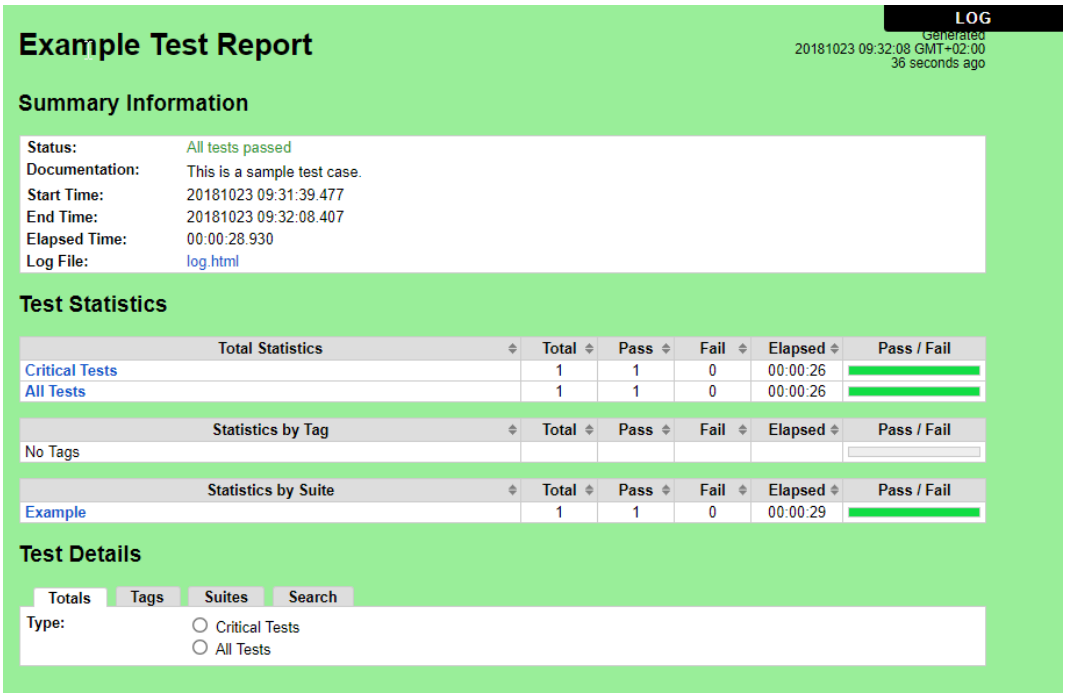


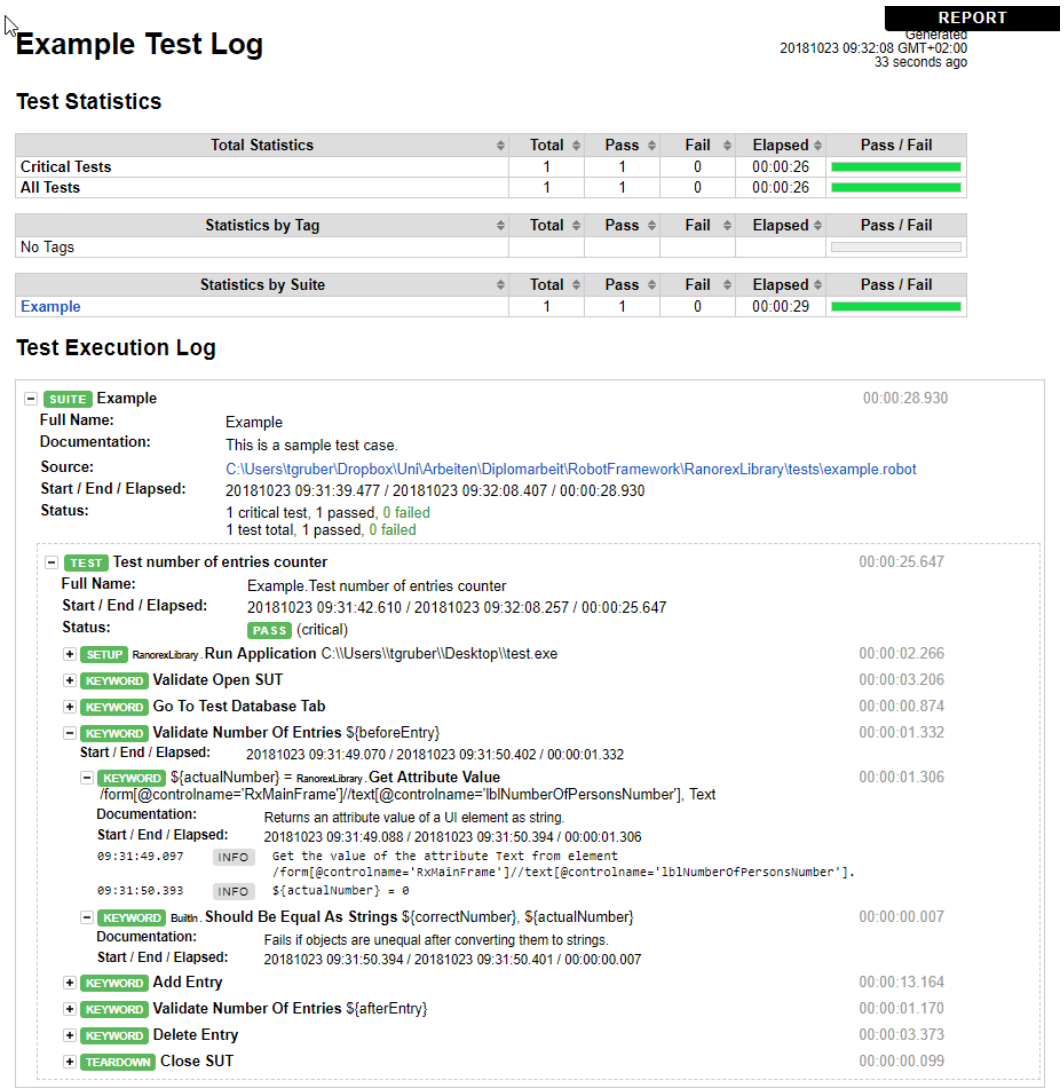
Figure 7.2: This screenshot of the Robot test report shows the main information about the test run. It shows the example test suite that took 29 seconds to complete, and some additional basic data like the start and end times of the test run. All the tests (just a single one in this case) are green.

```
Close SUT
    Close Application      /form[@controlname='RxMainFrame']
```

After saving the file it is sufficient to run this test from the command line using `ipy32 -m robot example.robot`. Now, Ranorex takes control over mouse and keyboard and we can watch the mouse cursor moving and key strokes appearing.

After the test has finished, we are presented with a report and a log file[8]. Both come as plain HTML files. A screenshot of the report is shown in figure 7.2 on page 62, and a screenshot of the log can be found in figure 7.3 on page 63.

[8]Robot also creates an `output.xml` file that collects all the output, if we create any.



Example Test Log

REPORT
Generated
20181023 09:32:08 GMT+02:00
33 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:26	
All Tests	1	1	0	00:00:26	

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

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Example	1	1	0	00:00:29	

Test Execution Log

SUITE Example 00:00:28.930
Full Name: Example
Documentation: This is a sample test case.
Source: C:\Users\tgruber\Dropbox\Uni\Arbeiten\Diplomarbeit\RobotFramework\RanorexLibrary\tests\example.robot
Start / End / Elapsed: 20181023 09:31:39.477 / 20181023 09:32:08.407 / 00:00:28.930
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

TEST Test number of entries counter 00:00:25.647
Full Name: Example.Test number of entries counter
Start / End / Elapsed: 20181023 09:31:42.610 / 20181023 09:32:08.257 / 00:00:25.647
Status: PASS (critical)

SETUP RanorexLibrary . Run Application C:\\Users\\tgruber\\Desktop\\test.exe 00:00:02.266
KEYWORD Validate Open SUT 00:00:03.206
KEYWORD Go To Test Database Tab 00:00:00.874
KEYWORD Validate Number Of Entries ${beforeEntry} 00:00:01.332
Start / End / Elapsed: 20181023 09:31:49.070 / 20181023 09:31:50.402 / 00:00:01.332

KEYWORD ${actualNumber} = RanorexLibrary . Get Attribute Value 00:00:01.306
/form[@controlname='RxMainFrame']//text[@controlname='lblNumberOfPersonsNumber'], Text
Documentation: Returns an attribute value of a UI element as string.
Start / End / Elapsed: 20181023 09:31:49.088 / 20181023 09:31:50.394 / 00:00:01.306
09:31:49.097 INFO Get the value of the attribute Text from element /form[@controlname='RxMainFrame']//text[@controlname='lblNumberOfPersonsNumber'].
09:31:50.393 INFO ${actualNumber} = 0

KEYWORD BuiltIn . Should Be Equal As Strings ${correctNumber}, ${actualNumber} 00:00:00.007
Documentation: Fails if objects are unequal after converting them to strings.
Start / End / Elapsed: 20181023 09:31:50.394 / 20181023 09:31:50.401 / 00:00:00.007

KEYWORD Add Entry 00:00:13.164
KEYWORD Validate Number Of Entries ${afterEntry} 00:00:01.170
KEYWORD Delete Entry 00:00:03.373
TEARDOWN Close SUT 00:00:00.099

Figure 7.3: This log shows much more detailed information about the test run. Every single keyword is listed with documentation, start and end time and additional information. If a keyword consists of other keywords, those can also be expanded and investigated. For debugging Robot tests, the log provides all the interesting information.

7.4 Documentation

The documentation of the RanorexLibrary can be found on the official GitHub[9] project. In order to keep everything in one place, this is the one resource for the RanorexLibrary. Since this project is supposed to grow and is subject to change, the documentation will also have to develop together with the code itself. Thus, including documentation in any other document doesn't reflect the dynamic nature of the project.

[9]Link to the official documentation: `https://github.com/Thomas-Gruber-90/RanorexLibrary/wiki/Documentation`

8 Conclusion

This thesis introduced the RanorexLibrary, a library for the Robot Framework that enables the tester to use the powerful object recognition of Ranorex directly in their Robot tests. This in itself already provides some benefits, especially since the implemented keywords are all on a low level and can be easily combined to more complex keywords. For example, clearing a textbox is not directly implemented in the RanorexLibrary, although this might be a common use case. To make this action work consistently, all characters in that textbox should get deleted, making it necessary to select all of them. This keyword could look like this (in Robot syntax):

```
| Clear Textbox |  |                |
|  | [Arguments]  | ${ranorexpath} |
|  | Double Click | ${ranorexpath} |
|  | Key Shortcut | {Delete}       |
```

There are many higher level keywords that could be possible and implemented within the RanorexLibrary. However, in this iteration I have focused on the lower level keywords that are directly mappable to a Ranorex action, as these actions are sufficient to do most of the important steps in a UI test.

Since Ranorex is a proprietary tool, the RanorexLibrary will only be interesting for a certain group of testers that either already use Ranorex or already use Robot and are willing to spend money on getting the Ranorex functionality. There are other solutions that can do similar things to Ranorex that are open-source, however, all these projects only support a single technology. If the tester wants to create a real end-to-end test that touches several different technologies, the RanorexLibrary is the only way to make this test easily maintainable and readable.

On the other hand, the RanorexLibrary itself is an open-source project. Thus, its future development can be geared towards fulfilling the needs of its actual users. Hosting it on GitHub means that the project can stay active as long as as someone wants to use it.

Bibliography

[1] M. Foord and C. Muirhead, *IronPython in Action*. Greenwich, CT, USA: Manning Publications Co., 2009, ISBN: 1933988339, 9781933988337 (cit. on pp. 1, 32, 34).

[2] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2012, ISBN: 978-1118031964 (cit. on pp. 9, 10).

[3] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, *et al.*, "Manifesto for agile software development," 2001 (cit. on p. 10).

[4] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 2005 (cit. on p. 11).

[5] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Proceedings of the 7th International Workshop on Automation of Software Test*, IEEE Press, 2012, pp. 36–42 (cit. on p. 12).

[6] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (gui) testing: Systematic mapping and repository," *Information and Software Technology*, vol. 55, no. 10, pp. 1679–1694, 2013, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.03.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913000669 (cit. on p. 12).

[7] P. Laukkanen, "Data-driven and keyword-driven test automation frameworks," Master's thesis, Helsinki University of Technology, Helsinki, Feb. 2006 (cit. on pp. 13, 16).

[8] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016, ISBN: 978-0-521-88038-1 (cit. on p. 14).

[9] S. Bisht, *Robot Framework Test Automation*. Birmingham, UK: Packt Publishing Ltd., 2013, ISBN: 978-1-78328-303-3 (cit. on p. 16).

[10] R. F. Foundation. (Aug. 29, 2018). Robot framework, [Online]. Available: http://robotframework.org (cit. on p. 16).

[11] P. Klärck. (Sep. 6, 2018). Experience, [Online]. Available: http://eliga.fi/experience.html (cit. on p. 16).

[12] Ranorex. (Sep. 5, 2018). Ranorex, [Online]. Available: https://www.ranorex.com (cit. on p. 21).

[13] *Ranorex sales demo slides*, As Sales demo for interested prospects, PDF can be received from either the Ranorex Sales team or the author. (cit. on p. 23).

[14] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2013, pp. 108–113. DOI: 10.1109/ICSTW.2013.19 (cit. on pp. 25, 55).

[15] M. Grechanik, Q. Xie, and C. Fu, "Creating gui testing tools using accessibility technologies," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, Apr. 2009, pp. 243–250. DOI: 10.1109/ICSTW.2009.31 (cit. on p. 28).