CHALMERS

Automatic Regression Testing using Visual GUI Tools

Master of Science Thesis in Computer Science: Algorithms, Languages and Logic

Johan Sjöblom and Caroline Strandberg

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 22, 2014

Automatic Regression Testing using Visual GUI Tools

JOHAN SJÖBLOM
CAROLINE STRANDBERG

Examiner: Gerardo Schneider
Supervisor: Emil Alégroth

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Cover: Giraffe AMB, used with courtesy of Saab AB.

This thesis has been prepared using LaTeX.

Department of Computer Science and Engineering
Gothenburg, Sweden, October 22, 2014

Abstract

Present-day software companies are facing growing demands of quality of their products but shorter delivery times, which affects the whole development process. The testing process, taking place at all levels of development, verifies that the product meets the requirements and expected quality. This process is often carried out manually, which is costly, time consuming and error-prone. Automated testing is proposed as a solution to this as it can raise the test frequency, giving faster feedback to the developers, and may run without human interaction. The advantages of automated testing is widely recognised, however the support for higher level – i.e. the graphical user interface (GUI) – system testing is limited. Automated testing becomes even more important when it comes to regression testing, since this will assure that the system fulfils the requirements after updates have been made.

This thesis aims to investigate the deployability of automating regression tests for a Command, Control and Communication system at Saab AB, using visual GUI testing. Visual GUI testing uses image recognition to find visible objects on the screen and has several advantages over previous techniques for GUI testing. It was invented in the early nineties, however due to the computational heaviness of the image recognition algorithms it was not until recently the hardware and software became powerful enough to make the technique usable in an industrial context.

The investigation performed consists of an evaluation of two existing visual GUI testing tools – Sikuli and JAutomate – which is done with respect to empirically measured performance and perceived usability based on the tools' specifications. Due to the complexity of the tested system, animated objects and fault tolerance also becomes important in the evaluation. The evaluation concludes with a comparison between the tools, as well as a recommendation to Saab AB of which is most suitable for the tested system. Sikuli was chosen due to the expressivity and powerfulness of the scripting language.

The text recognition algorithms worked poorly, which was improved by implementing a training function for the text recognition in Sikuli. This improvement was then evaluated by comparing the results from running the original text recognition and the trained text recognition.

The research on animated interfaces and visual GUI testing is limited, making our contribution in this aspect valuable. We can conclude that both of the tools can handle this; however the fault tolerance of the scripts becomes crucial. A topic for future work would be to investigate the long-term costs for development and maintenance. However, so far visual GUI testing have been found to be a very promising technique that work well and have a lot of potential.

Acknowledgements

First, we would like to thank our supervisor at Saab AB, subdivision Electronic Defence Systems in Gothenburg, Martin Karsberg for his help and support throughout the project, and for coming up with the Master's thesis proposal and giving us the opportunity of performing this at Saab AB.

Next, we would like to thank Team Oden at Saab AB, subdivision Electronic Defence Systems in Gothenburg, for welcoming us into their Scrum team and for their help and support of the tested system throughout the project. A special thanks to Tommy Bengtsson for helping us with the found bugs in the system and to Christoffer Höglund for helping us with our final demonstration and presentation of the work at Saab AB.

We would also like to express our deepest gratitude to Vilhelm Bergman at Saab AB, subdivision Electronic Defence Systems in Gothenburg, for the extensive the help and support he has given us during the project; both with supporting the tested system and selecting representative test cases, and with the report.

Finally, we would like to thank our supervisors at Chalmers University of Technology and University of Gothenburg Emil Alégroth and Gerardo Schneider for their help and feedback throughout the project.

Johan Sjöblom and Caroline Strandberg, Gothenburg 2014-10-22

Contents

1 Introduction

This thesis is performed in collaboration with Chalmers University of Technology and Saab AB, subdivision Electronic Defence Systems. The goal for this thesis is to investigate the ability to automate regression tests for graphical user interfaces (GUI's) of a Command, Control and Communication system using visual GUI testing tools. The evaluation of the tools is done with respect to empirically measured performance, perceived usability based on the tools' specifications, and perceived cost and time saved.

1.1 Background

Software testing is an important part of the software development process to ensure product quality. Today, all testing on graphical user interfaces (GUI's) at Saab AB is done manually, which is slow and expensive. Thus, many regression tests are rarely run and often only a subset of all test cases is used. A large part of the lower levels of the systems are automatically tested – for example by unit testing the code – and Saab AB sees great benefits in expanding the automatic testing to include graphical interfaces as well. Testing is a big and important part of software development, thus automating it means the tests can be run more frequently and hopefully at a lower cost than manual tests [1].

Visual GUI testing is chosen as it is a promising technique for automating high level test cases in a system, and it avoids some of the drawbacks of other similar techniques, such as widget based GUI tools [2], which is sensitive to GUI layout changes or changes in software design and code – resulting in high maintenance costs. The goal is to expand the parts of the system being tested automatically, rather than just replacing manual tests with visual GUI testing.

Manually running a test suite could take up to several man-hours if it is big, thus of course burdening the organisation with a large cost. Because of this, tests are usually carried out less often than what would be desirable. As a contrast, visual GUI testing can be run often with next to zero cost for each run. However, the script to be run has to be developed, which is a process that also usually takes several man-hours, which is associated with a development cost. Of course, once written, the script can be used many times. When and if the initial cost of development is outweighed by the time saved by repeatedly running the tests, depends on their complexity and how often these parts of the system needs to be tested.

When performing testing, organisations often have structured documents that dictate how the testing is to be performed and the expected outcome of the test. These documents will be referred to as “regression test verification specifications” in this report. These documents are associated with a maintenance cost, since they need to be kept up to date with the system itself, once the latter is updated. In the same way, the scripts of the visual GUI testing tool also need to be kept up to date and also have a maintenance cost. These scripts may need some maintenance if something in the system has changed, for instance if some GUI parts are replaced.

1.2 Scope of work

This thesis aims to investigate the ability to automate regression tests for GUI’s of a Command, Control and Communication (C3) system at Saab AB. By automating regression tests, the testing can be more effective and run more frequently while the manual testing can focus on exploratory testing. The goal of this thesis is to evaluate the robustness of current state-of-practice visual GUI testing tools, as well as improving the Optical Character Recognition (OCR) functionality for one of the evaluated tools due to poor results of the text recognition.

The automation is done using a visual GUI testing tool, which uses image recognition to find objects in the interface. Since there are several such tools on the market, a comparison between two such tools – Sikuli and JAutomate – is done to find the most suitable. The number of such tools on the market is limited, and these were chosen on request from Saab AB. Another such tool that could have been tested is eggPlant [3] The evaluation and comparison is qualitative and quantitative and the evaluation method is further described in Section 5.1. A representative subset of Saab AB’s GUI test cases is implemented and tested using these tools. The result is evaluated with respect to empirically measured performance, perceived usability based on the tools’ specifications, and finally on perceived cost and time saved. In the evaluation, focus will be on how well animated objects are handled as well as the ability to achieve fault tolerance in the scripts of the tools. The tested system is complex, and includes communication between two units that are to be synchronised. The complexity of the system makes the ability to handle moving objects and fault tolerance crucial. A technical evaluation of the robustness of using visual GUI testing on animated objects is given through the experimental results which also show the importance of fault tolerance.

An important part of image recognition is recognising text. The tested tools both include OCR functionality, however, the accuracy of it turned out to be low, leading to the decision of improving it by adding a training function. Thus, the work also includes an implemented training function for improving the results of the OCR functionality in Sikuli and an evaluation of the improvement where the trained OCR engine is compared to the original untrained OCR engine. Finally, a recommendation of a tool and a set of recommendations are given to Saab AB for their future decision-making regarding this technology.

The tool evaluation result in that Sikuli is considered more suitable than JAutomate, mostly because of the script language expressivity. Some problems are found when handling moving objects as well as some ways of mitigating them, which shows the importance of fault tolerance. The thesis makes the following contributions:

- A technical evaluation of the robustness of using visual GUI testing on moving objects, also demonstrating strengths and weaknesses, and how problems can be overcome.
- Experimental results showing the importance of fault tolerance concerning visual GUI testing with moving objects.
- An implemented training functionality for improving the results of the OCR functionality in Sikuli, using Tesseract, together with a technical evaluation of the improvement.
- Presentation of five different faults that may appear using a visual GUI testing tool on a system with moving objects.
- A recommendation of a suitable visual GUI testing tool for Saab AB.

In our work, we do not analyse the maintenance cost of the Visual GUI Testing technique and the development cost is just briefly discussed. The technology is still novel and limited research has been conducted on the long-term costs of the technique. We do acknowledge the importance of the maintenance cost of a new technology; for example, the widget based GUI tools were very promising when introduced, but the maintenance costs of the scripts are perhaps their biggest limitations [2, 4]. The tested system as well as the tools are adapted to minimise external interference and maximise the performance.

The thesis is structured as follows: In this chapter, the work with a background and purpose for the research has been introduced. This is followed by some theory about Radar systems in general (Chapter 2) and automated testing (Chapter 3). The test environment is then described (Chapter 4), after which the evaluation, discussion and comparison of the tools is presented, which results in the selection of a suitable tool (Chapter 5). This is followed by describing the implementations performed during this thesis (Chapter 6). The thesis ends with presenting our results (Chapter 7), followed by a discussion (Chapter 8), future work and some conclusions drawn from this work (Chapter 9).

1.3 Research methodology

The methodology used in this thesis is divided into three parts: first a pre-study, then an industrial study at Saab AB which resulted in a comparison and tool recommendation, and finally implementing an extension of the recommended tool.

The pre-study consisted of collecting knowledge and information from literature and scientific papers about the chosen visual GUI testing tools, Sikuli and JAutomate, aimed to be used as a part of an evaluation of the tools. This included some prior testing of the tools as well, to make the later implementation steps easier. The pre-study also included collecting knowledge and information about both automated testing and visual GUI testing, and the techniques behind it. Another important part of the pre-study was to get familiar with the tested system at Saab AB and chose some regression tests that should be implemented during the industrial study, in order to test the tools and prove the concept of visual GUI testing. This was done by informal interviews of testers and developers at Saab AB.

The industrial study consisted of us implementing the chosen regression tests from the pre-study to test the tools' performance and usability on the system. By conducting experiments using the tools and running the implemented regression tests, quantitative data regarding the tools' strengths and weaknesses were collected. This also provided information of how well moving objects were handled, which is an important part of the tested system. The results from the pre-study of the tools and the observed results from the tests that were run, was used as data in an evaluation of the chosen tools – which is further described in Section 5.1 – and resulted in a comparison where one tool was recommended as most suitable for the tested system.

Due to the poor results regarding the OCR functionality of the chosen tool that was experienced during the industrial study, an investigation about improving the accuracy for this functionality was performed. This resulted in that we found that a training function could be used for improving the performance of the OCR functionality. Thus, a training function was implemented as an extension to the chosen tool with the goal of improving the results. The performance of the modified OCR functionality was measured by conducting tests using the tool both with and without the modification. This resulted in that quantitative data was collected for the performance of the OCR functionality, which was evaluated in order to determine the improvement.

The following research questions are addressed:

- Is it possible to use image recognition (by using visual GUI testing) to analyse and test that the interfaces reflect the underlying data? How robust are these tools and what are the limitations?
- Does moving objects pose problems? If so, can they be handled, and how?
- What kind of faults are possible to get in an animated interface using a visual GUI testing tool?

- Is it possible to improve the accuracy of the OCR functionality in Sikuli, perhaps by a training function? If so, how much improvement is possible? What are the limitations of such a training function?
- What are the perceived costs relative to the benefits?

Based on previous similarly studies performed regarding visual GUI testing, as described in [2], the hypothesis was that visual GUI testing and image recognition will work well as a way to analyse and test GUI's. We also thought that moving objects might pose a problem and thus might be a limitation. The modified OCR functionality was expected to significantly improve the results.

2

Radar systems

Radar is an acronym of RAdio Detection And Ranging, and the technology is used for detection and range estimation of objects using radio waves [5]. The first to use radio waves for detection of objects was the German inventor Christian Hülsmeyer in 1904 [6]. The radar technology in the modern sense, however, was invented in the 1930s by the British for defence against military aircraft. Today, it is widely used for both military as well as civilian purposes. A radar system emits electromagnetic pulses, i.e. radio waves, to detect and localise reflecting objects [7]. It has the ability to determine for instance the range, speed, altitude and direction of objects. Some radar types are also able to determine the shape and size of an object, and can distinguish between different types of targets, thus classifying the object [7]. A reflecting object that a radar system can detect could be for instance aircraft, ships, missiles or terrain [7].

2.1 Radar basics

A radar system has one antenna for transmitting and one for receiving, and they are usually combined into one physical antenna. The transmitting antenna will emit an electromagnetic pulse of some frequency, which varies between different types of radar systems. When the pulse reaches the target and hits it, the signal is reflected [5]. The echo from the reflecting target can then be received by the antenna. This process is shown in Figure 2.1.

Electromagnetic energy travels with the speed of light. Thus, the location of the reflecting object can easily be calculated using the time difference between the emitted pulse and the received echo [5]. By doing this several times, information like velocity can be calculated and a direction can be predicted. Since a radar system only uses the reflection of its own signal and no other radiation sources, and it is independent if it is day or night.

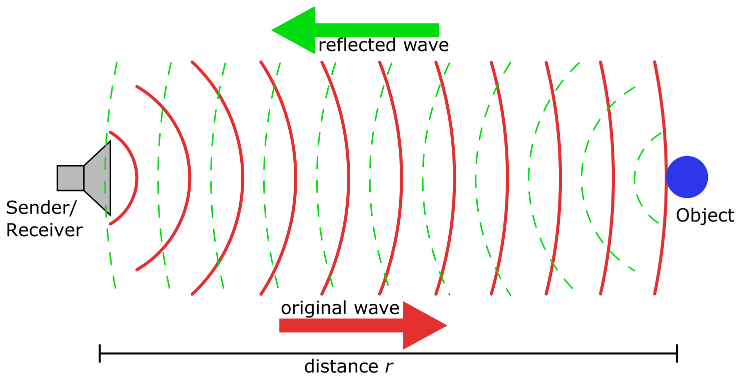


Figure 2.1: The radar transmitting/receiving process [8].

The power, P_r, received by the receiving antenna – if the transmitter and receiver are placed together – is given by the radar equation (Equation 2.1)

$$P_r = \frac{P_t G_t A_r \sigma F^4}{(4\pi)^2 R^4}, \tag{2.1}$$

where P_t is the transmitted power, G_t is the gain of the transmitter, A_r is the effective area of the receiver, σ is the cross section of the target, F is the pattern propagation factor and R is the range to the target. Important to note in this equation is that the receiver power is strongly decreasing with the range of the target.

Radars systems are an important technology, especially in the military, for uses such as air defence [7]. Radar technology is often used for surveillance, both in air as well as over land or sea. Radars systems can also be used for weather observations and air traffic control [7].

2.2 C3 systems

From Equation 2.1, it can be easily seen that the further away the target is, the more difficult it will be to detect. This can be compensated by having many radar units at different locations. Data can be gathered from all of them, and be combined into one view. In a military system, this is referred to as a Command and Control system (C2) or a Command, Control and Communication system (C3) for a land-based radar system, used for surveillance and air defence. This thesis will focus on C3 systems. A C3 system is an information system used in the military [9]. In general, it includes both strategic and tactical systems; for instance a combat direction system, tactical data system or a warning and control system. Worth noting is that a C3 system is a human function as well [9].

A Command and Control (C2) system is a subset of a C3 system. The U.S. Department of Defense Dictionary of Military and Associated Terms define Command and Control (C2) as follows:

> *"The exercise of authority and direction by a properly designated commander over assigned and attached forces in the accomplishment of the mission. Command and control functions are performed through an arrangement of personnel, equipment, communications, facilities, and procedures employed by a commander in planning, directing, coordinating, and controlling forces and operations in the accomplishment of the mission. Also called C2."* [10]

Their definition of a Command and Control system (C3) is:

> *"The facilities, equipment, communications, procedures and personnel essential to a commander for planning, directing and controlling operations of assigned forces pursuant to the mission assigned."* [9, 10]

The C3 system assists the human in command in the decision-making process and provides the military staff with information, where the information flow to the commander is prioritised. The information flow is based on sensor data. Thus, a greater coverage can be received by integrating the system into a sensor net, i.e. having several radar units. This communication system is heavily used in military operations, for instance for regrouping units on the field, for protection and warning against enemies or for protection of friends to avoid fratricide. It can also be used for determining whether or not to fire at a hostile unit and for tracking missiles.

The tested system at Saab AB is an interface to the tactical control of a C3 system, used in a Giraffe AMB (shown on front page) which is used for ground based air defence. Such a system is also shown in Figure 2.2, which shows a C3 system with two persons in a Giraffe AMB. Each operator has two displays showing the radar data for their radar unit and the displays for the two persons are identical.

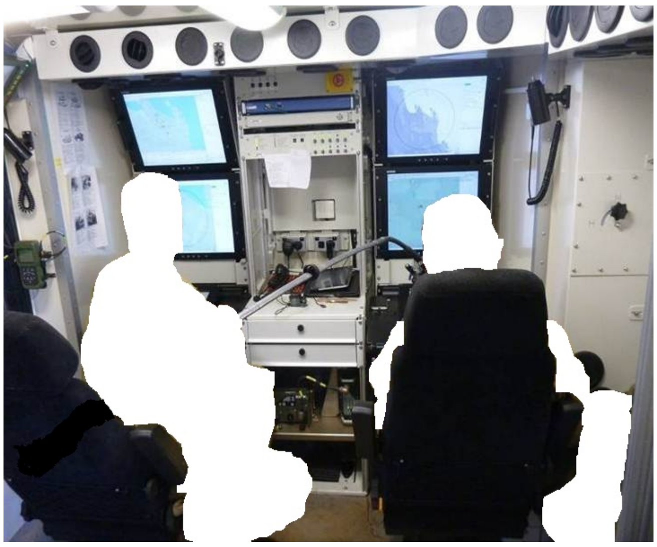


Figure 2.2: Two operating centres for a C3 system in a Giraffe AMB. The persons have been retouched out for security reasons. The figure is used with courtesy of Saab AB.

In the tested interface (i.e. the same as that used on the C3 system in Figure 2.2), detected objects are displayed and can be classified. Objects are shared between operators and different units. The concept of shared data between different units is described as a common air picture, which is shown in Figure 2.3.

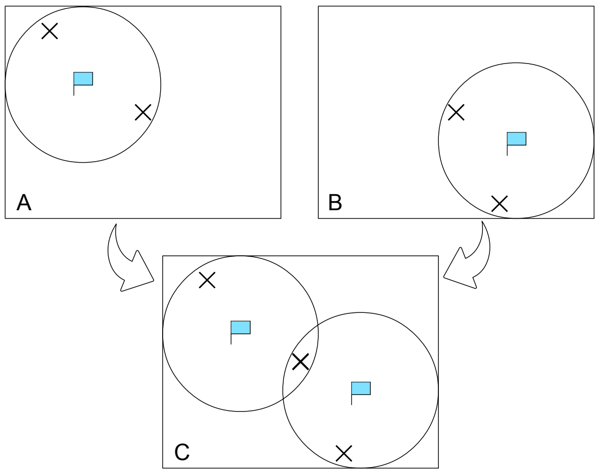


Figure 2.3: Common air picture. Two radar systems, A and B, are shown. There are three objects in the system, and A and B detects two objects each, with one object being seen by both radars. By sharing the data between each other, the common air picture shown in C can be used for both of the systems.

Figure 2.3 shows two different radar units – A and B – that are stationed at different places and thus detects different objects, marked with X. These two units share their data with each other which results in the combined air picture C which is the union of the radar data for the two systems. This results in that both system A and system B see the radar view shown in C, even though they alone do not detect all objects shown in C. Also note that the view in C only has three objects, since the middle one is the same physical object detected by both A and B. The tested system is further described in Chapter 4.

3

Automated Testing

Software companies of today are often faced with growing demands of quality while delivery time gets shorter. This affects the entire process from architecture to implementation. The testing process, which takes place at all levels of development, verifies that the product meets the requirements and expected quality. This process is often carried out manually, which is both costly and time consuming. Automated testing is proposed as a solution to this as it can raise the test frequency, giving faster feedback to the developers, and may run without human interaction [1]. The advantages of automated testing makes it widely used among software development companies and more and more of the testing gets automated. The automated testing technique contains everything from unit testing to widget based GUI tools; however among these techniques the support for higher level acceptance testing is limited.

The automated testing technique becomes even more useful when it comes to regression testing [11]. Regression testing is used to verify that a system continues to meet the requirements after a change has been made in the system. The regression tests for a system consist of a prioritised subset of all tests for the system which often test longer chains of actions. The ideal case would be to test the system every time a change has been made. However, since manual testing is a slow and expensive technique, it is often performed on an irregular basis which is usually done in connection with deliveries. This can make the system error-prone since some errors do not appear very often and thus demands more extensive testing to be found. By using automated testing for regression tests, these tests could be run much more often, for example every night. This increases the possibilities of finding erroneous behaviour and therefore improves the quality of the product and shortens delivery time. This is also a matter of confidence for the developers, since each change in a system could theoretically break something in the system, which makes people a bit careful. Regression tests can show that the system still works as expected after a change has been made.

Automatic testing cannot replace manual testing, but should rather be seen as a complement [12]. An important approach to manual testing is exploratory testing, where the tester uses his or her experience along with creativity to come up with new tests [12]. Unless this learning and creativity can be modelled by a program, exploratory testing should still be conducted by human testers. However, it may be possible to extend the automated testing even further in the future by using random visual GUI testing [13], and is to some extent possible already today by randomly interacting with the tested system and verify that the system does not crash.

3.1 Automated GUI Testing

Most automated testing techniques consider lower levels of the system, for instance unit testing, but there are some techniques aimed at the higher levels, i.e. the graphical user interface (GUI) of a system, such as widget based GUI tools or Visual GUI Testing (which will be introduced in Section 3.3 and is the focus of this thesis study) [14]. These techniques can execute tests close to how a human tester would, allowing testing of functionality and performance.

3.1.1 Widget based GUI tools

There are two different generations of widget based GUI tools, coordinate-based and GUI component-based. Widget based GUI tools works in two steps, first the user's interaction in the system is stored using a recording tool. Then the widget based GUI tool plays the recording, and automatically interacts with the system the same way the user did [2].

The coordinate-based widget based GUI tools are sensitive to GUI layout changes since it depends on static x and y coordinates [2]. Thus, moving anything in the GUI may cause the script to fail. The result is high maintenance cost, making it unsuitable for regression testing [4]. However, it is not affected by changes in software design or code.

The GUI component-based approach uses the properties of the GUI components instead of the coordinates which makes it more robust to changes in the layout [2]. However, it is sensitive to changes in software design and code, which could cause a whole script to fail if for example the Java version is changed [15]. This method also has the drawback that it may need a tool to access the source code to get the components properties, which may not work when components are generated during runtime. Besides, a user would not be able to find and interact with hidden objects, in contrast to the widget based GUI tools. This is because the method interacts using the components' internal functions, such as the click function, rather than that function of the operating system [2].

3.2 Image recognition

Image recognition is a kind of computer vision and the core technique used in visual GUI testing. The goal of image recognition is to make a computer understand an image, in the same way a human would. Image recognition can be seen as trying to find symbolic data from an image by using geometry, statistics, physics and learning algorithms [16]. A classical field of application for image recognition is to determine if an image contains a specific object, such as a geometric object, a human face or a character.

There are a number of different image recognition techniques that work in different ways. One approach is to use a machine learning algorithm to teach the computer to recognise some pattern. This is done by "feeding" the algorithm with data consisting of different kinds of images, which the computer uses for learning to distinguish between the types of patterns in the data [16]. The knowledge that the computer gains during this process can then be used to find those patterns in other images, i.e. the computer has "learned" to recognise some pattern, for instance a specific image.

The learning process works in a similar way to how humans learn, by giving the algorithm some data and the expected classification. The algorithm then classifies the given data by simply guessing. If the algorithm does not give the expected classification, it is automatically corrected. This procedure should then be repeated to assure that the expected result is acquired. Such a learning process is rather similar to humans learning vocals. An example of such a machine learning algorithm is Bayes classifier and the k-nearest neighbours algorithm. Another example is the Perceptron algorithm shown in pseudo code in Algorithm 1. By using this for, if possible, several similar images, it is possible to learn approximately what an image looks like and be able to recognise similar images, despite there being some minor differences. This works because the image recognition algorithms do not look for an exact match, just something similar to the sought image.

The Perceptron algorithm is a binary classifier, which means that it tries to classify the data into two classes by fitting a straight line (i.e. $y = kx + m$) between them, using the weight vector as coefficients. The length of the weight vector is the same as the dimension of the training data, e.g. training data in a space with x-, y- and z-coordinates, gives a classification by fitting a plane in the space to separate the data set. `sign(v)` is the signum function, which gives 1 if $v > 0$ and -1 if $v < 0$. An example of a data set classified using a Perceptron algorithm is shown in Figure 3.1.

Another approach is given by noting that image recognition is correlated to image segmentation, which is used to partition an image into segments such that pixels with certain characteristics are grouped together [21]. Then cross-correlation can be used to find an object in an image. Cross-correlation has its roots in signal processing and the idea is to use a small image and find it in a bigger image. The cross-correlation is defined in Equation 3.1 as

$$(f \star g)(t) = \int_{-\infty}^{\infty} \overline{f}(-\tau)\ g(t-\tau)d\tau, \tag{3.1}$$

where f and g are continuous functions and $\overline{f}(t)$ is the complex conjugate of $f(t)$ [22].

Algorithm 1: Perceptron algorithm, interpreted and written based on the algorithm descriptions in [17, 18, 19].

```
   Data: Xtrain: training data, Ytrain: labels of training data
   Result: Weight vector $w$
1  Initialise weight vector $w$ to all zeros
2  $N \leftarrow$ number of iterations through training set
3  for $1 \rightarrow N$ do
4      forall the $x$ in Xtrain, $y$ in Ytrain do
5          $guess \leftarrow sign(x \cdot w)$
6          if $guess \neq y$ then
7              $w \leftarrow w + (y \cdot x)$
8          end
9      end
10 end
11 return $w$
```

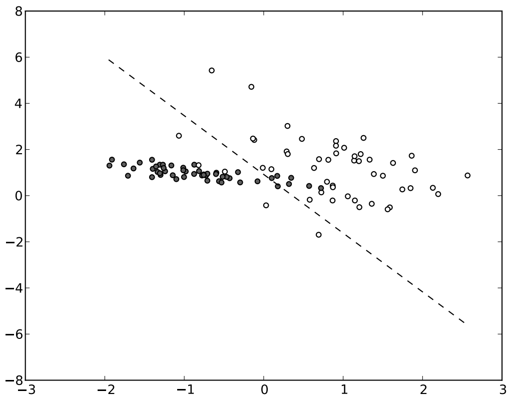


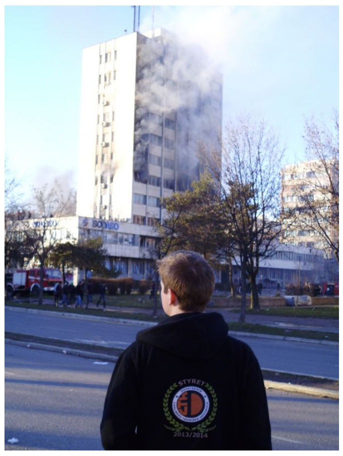
Figure 3.1: Example of an output from a Perceptron algorithm [20].

For discrete functions, the integral in Equation 3.1 becomes a sum, and for two images – represented as two matrices A and B – of different size, the cross-correlations is in Equation 3.2 defined as

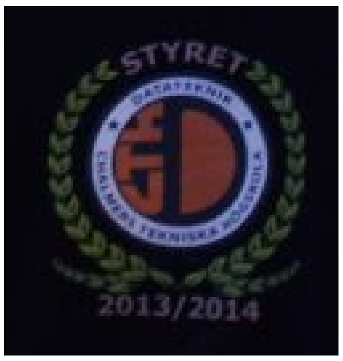
$$C(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A(m,n)\overline{B}(m-i,n-j), \qquad \begin{array}{l} -(P-1) \leq i \leq M-1, \\ -(Q-1) \leq j \leq N-1 \end{array} \tag{3.2}$$

where A is a matrix of size $M \times N$, B is a matrix of size $P \times Q$ and C is a matrix of size $(M+P-1) \times (N+Q-1)$ [23].

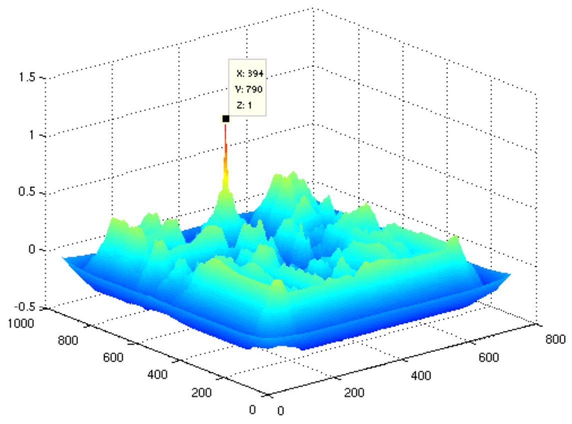
By using cross-correlation, which calculates the sum of the functions' product at each position (pixel by pixel), a matching is given by a maximisation since aligned peaks makes a large contribution to the sum. An example of this can be seen in Figure 3.2.



(a) Picture from the Bosnian Spring [24].



(b) Detail of Figure 3.2a.



(c) Plot of how well Figure 3.2b fits into Figure 3.2a.

Figure 3.2: Image fitting by cross-correlation. Figure 3.2c shows the result from the cross-correlation of Figure 3.2a and Figure 3.2b, which shows how well Figure 3.2b fits in different parts of Figure 3.2a. The clear peak that is marked shows where the best match was found. A thin peak indicates a very clear match while a thicker "mountain shape" indicates that the smaller figure did not match as well.

3.2.1 Optical Character Recognition

Optical Character Recognition (OCR) is a kind of pattern recognition and computer vision software, just like image recognition, which allows conversion of images containing text (usually scanned) into text or information understood by a computer [16, 25]. OCR was originally a technique for converting machine-printed text on a paper into an electronic form and is still used world-wide to digitise printed text, for instance passports and articles, so that the text for instance can be edited or searched [26].

The OCR technique is complicated, and it is important that the translation of the text from the image is correct. If an OCR algorithm makes many mistakes, it is usually not useful at all, and it is often more efficient to manually type the text. Consider a page consisting of about 3000 characters. With a translation accuracy of 99%, there will be 30 errors. An example of what such a mistake can look like is shown by Rice, Nagy and Nartker [26]:

> *"What if an OCR system cannot reliably distinguish the letter **c** from the letter **e**? **You may cnd up with a scntcnec that looks likc this.** Imagine an entire page of such gibberish."*

OCR software, like for instance Tesseract, uses a machine learning approach similar to image recognition to classify the characters. The classifier of Tesseract is an optimised k-nearest neighbours classifier [27], as described in Section 3.2. Since it uses machine learning it is possible to train the software on new fonts or languages, by feeding it with new training data. Tesseract uses a two-pass system where the second pass learns from the first pass [28]. This is done by using the satisfactory words recognised in the first pass to recognise words that were not satisfactorily recognised in the first pass [28].

3.3 Visual GUI Testing

Visual GUI Testing (VGT) is a rather new technique even though the core of the technique was invented in the early nineties [29]. However, due to the computational heaviness of the image recognition algorithms used, it was not until recently that hardware and software have been developed to be powerful enough to make this technique usable for industrial applications [14]. Visual GUI testing uses image recognition and scripting to interact with the top GUI bitmap and input layer displayed to the user on the monitor of the tested system [30, 31].

Two different studies concerning visual GUI testing are presented in [2], which concludes that visual GUI testing is both gainful and beneficial.

3.3.1 Motivation

Previous automated GUI testing tools have had many drawbacks affecting their usability, as mentioned in Section 3.1.1. The main difference between visual GUI testing and widget based GUI tools is the image recognition algorithms operating on the highest abstraction level, making it unaffected by changes in the GUI layout, code or API [32]. It can interact with any type of component, e.g. images, buttons, text fields or custom components [32]. Visual GUI testing is also able to interact with animated GUI's which is not possible with previous techniques, such as widget based GUI. As the interaction is performed the same way as a human user would, the visual GUI testing technique works regardless of the tested system's operating system, architecture or implementation language.

3.3.2 Limitations

The main limitation of this technique is that visual GUI testing tools may fail to find the images being sought. Image recognition threshold parameters often need to be tuned in order to get accurate results. Many image recognition algorithms use statistical methods when classifying [33], making the classification non-deterministic. Thus, while the accuracy may be high, a script could fail during the execution. For instance, Sikuli uses supervised machine learning algorithms that are based on statistics [34]. This results in that the image recognition may sometimes fail, and a different number of features may be found on different runs for the same input. This is of course a problem when using this technique, since results cannot be completely trusted, thus demanding more of the developer to create intelligent scripts that handles the possible failures that may occur.

VGT tools are sensitive to changes in the bitmap components. This means that changing an objects colour or size may get the script to fail.

Like every other GUI-based technique, such as widget based GUI tools, the performance of visual GUI testing is affected by the performance of the system. The visual GUI testing scripts cannot run faster than the system can respond, which means that the scripts must be synchronised with the system. This can be done using delays in the testing scripts, but it also means that the user could get a feeling of the script being slow when in fact it is merely waiting for the system.

Little research has been conducted on animated GUI's [2], thus this could be a limitation.

3.4 Script execution states

If each action that one can take on a system is seen as moving from one state of the system to another, then a test script execution can be regarded as entering different states. Due to the uncertainty of VGT, as mentioned in Section 3.3.2, the test results can never be completely trusted. The tests therefore need to be of a quantitative nature; the more times a test is repeated, any failures in the VGT tools can be made arbitrarily small. Thus, the implemented scripts run a test, reset the system if needed, and then run the test again indefinitely. Each script execution is referred to as an iteration of the test. In Figure 3.3, a script execution is shown, where the system moves between states in the test script execution, whereupon the system is reset and the script will be run again.

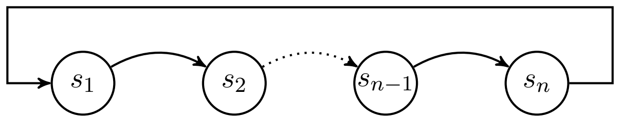


Figure 3.3: The states of the testing iteration loop. The test consists of the different steps s_1 through s_n to take. After all steps have been executed, the test iteration is done, whereupon the system is reset if needed and the next test iteration is carried out.

In the tested system at Saab AB, the complexity and the moving objects as well as the inherit uncertainty of the VGT tools meant that any action taken on the system could fail. Upon failure, trying to repeat the action would usually succeed. An example of where this could be applicable is if the script was about to click on an object that had moved from the place where it was detected, which meant the script would miss it. Figure 3.4 shows one iteration of the script execution, with the possible progressions between the states in the iteration. If the VGT tool detects the expected result – True Positive (TP)– the system moves to the next state. If the VGT tool detects that the result is not what was expected, it tries to re-perform the action – False Positive or True Negative (FP, TN). Note that the state needs to cope with the fact that the actions executed might have been performed correctly, but the VGT tool fails to see the expected result despite it being present – False Negative (FN). In other words, the normal procedure in state s_i would be to perform the action of that state expecting that the system came from state s_{i-1}, but it must also be able to handle that the system already is in state s_i. The possibility that the VGT tool detects the expected result despite it not being present (FP) is also possible. Guards and checks can be placed in the script to try to check that the state action succeeded, but these checks themselves can fail. The system is then regarded as ending up in an undefined state, since the VGT tool thinks that actions have been performed correctly. If this happens, subsequent state actions for the following states will be executed, despite the system not being in a well-defined state. This will cause follow up errors, but as long as the system is reset properly after an iteration, the follow up errors would be finite.

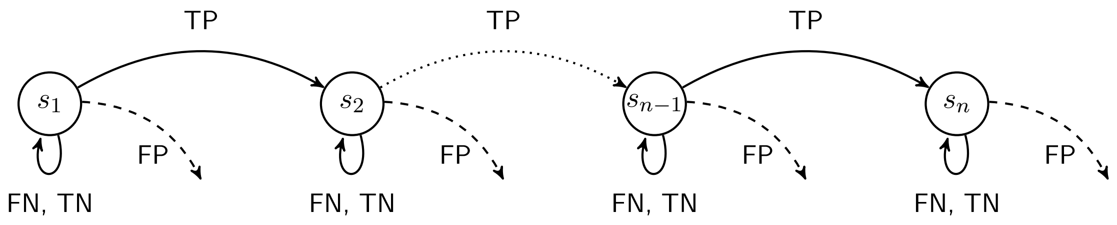


Figure 3.4: Possible paths through a single iteration. TN stands for True Negative i.e. the expected result is not present and the VGT tool correctly thinks that it is not present. FP stands for False Positive i.e. the expected result is not present but the VGT tool incorrectly thinks it is. FN stands for False Negative i.e. the expected result is present but the VGT tool incorrectly thinks it is not. TP stands for True Positive, i.e. the expected result is present and the VGT tool correctly thinks it is.

To exemplify these models, consider the action of setting the type of an object in the tested system. This is done by selecting an object, right-clicking on it in order to bring up its context menu, opening up the sub-menu for setting types, and then selecting the type wanted. The states that would be passed through are as follows. If the object is not visible on the screen, the system was improperly set up relative to the script, and the user would need to make sure the object is visible. If the object is present on the screen, as should be the case, this should result in a TP. The script should click on the object in order to select it, and immediately afterwards right-click on the object. Both of these clicks could have missed the object, resulting in that the expected context menu would not be present. This result in an FN, and the action would be performed again. In case the context menu does appear, but the VGT tool fails to find it (TN), the action would also be performed again. If the context menu is found, this results in a TP, and the script moves on. The script is now to open the sub-menu and select the appropriate type from the list. This could fail if the VGT tool fails to find the menu despite it being present (TN), or if the sub-menu does not appear (FN). In these cases, the script would reattempt the action. If the assignment went as expected (TP), the next state would be to verify that the assignment worked and possibly log any errors.

4

Test environment

THE TESTED SYSTEM CONSISTS of two simulated radar units, connected in a subnet. This is shown in Figure 4.1, where the two radar units are represented as dotted boxes. The radar units are Giraffe AMB's, which is a radar unit developed by Saab AB and used for ground based air defence [35]. The radar units are shown in the bottom right corner of each box. Thus, the tested system consists of two different subsystems (one for each radar unit; here referred to as system 1 and system 2), that communicate with each other. System 1 is illustrated as the right part of Figure 4.1, and system 2 is illustrated as the left part. Each system has a data processor (DP) which holds all radar data, and stores it in a database. The radar data is shared between the two systems. Each system also has one or two operating centres (OPC) which shows the tested interface. The radar data is simulated from one of the operating centres, which sends the radar data to all other operating centres within the subnet. One of these operating centres runs the application for the VGT tool. The computer running the VGT tool has multiple screens and is remotely connected, by Windows Remote Desktop connection, to the other operating centres in order to be able to see their screens as well. OPC1 of system 2 is connected to OPC1 of system 1 through the connection named "Remote Desktop" in the figure. OPC2 is connected through the switch since OPC1 and OPC2 of system 1 belong to the same subnet. Thus, the VGT tool can operate on all the running interfaces of the different OPC's through the Windows Remote Desktop connection.

As stated above, the radar data that is used is computer generated data and not real radar data. The system works fine with real radar data as well, but the tests in this thesis will only be performed with simulated data since it is much more predictable and well-defined.

The properties of system 1 respectively system 2 are presented in Table 4.1 and Table 4.2.

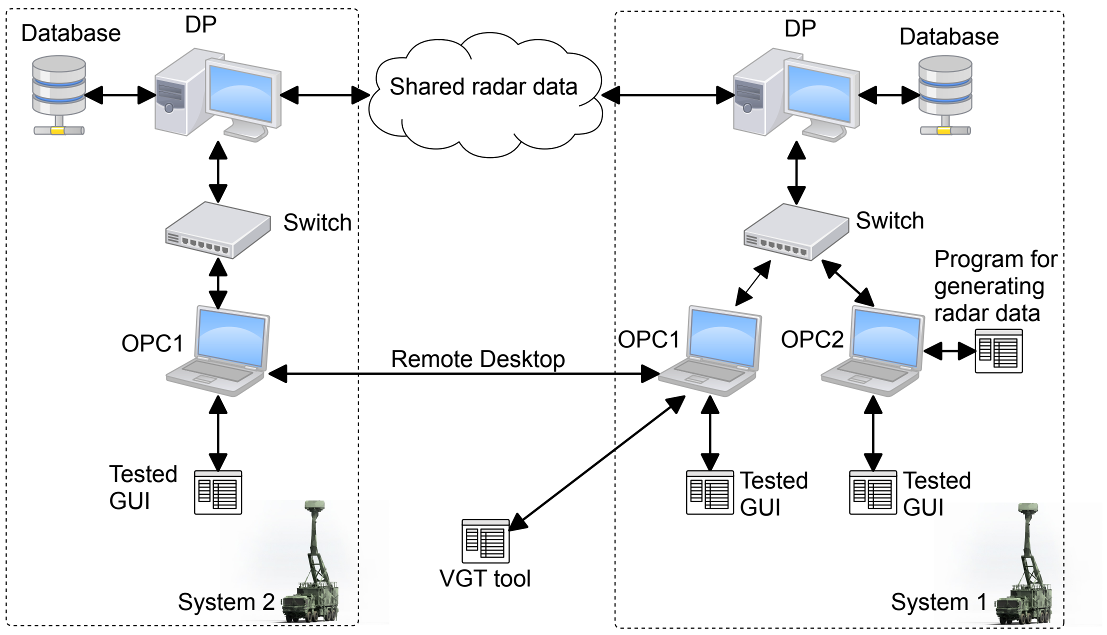


Figure 4.1: Illustration of the two systems, connected in a subnet.

	DP1	**OPC1**	**OPC2**
CPU Type	Intel Core 2 Duo E4500	Intel Core 2 Duo T7250	Intel Core 2 Duo T7500
CPU Frequency	2,20 GHz	2,00 GHz	2,20 GHz
Memory (RAM)	1,80 GB	3,00 GB	2,00 GB
OS	CentOS 6.5 (64-bit)	Windows 7 Professional (32-bit)	Windows 7 Professional (32-bit)

Table 4.1: Properties of System 1

	DP1	**OPC1**
CPU Type	Intel Core 2 Duo E6550	Intel Core 2 Duo T7250
CPU Frequency	2,33 GHz	2,00 GHz
Memory (RAM)	1,90 GB	2,00 GB
OS	CentOS 6.5 (64-bit)	Windows 7 Professional (32-bit)

Table 4.2: Properties of System 2

4.1 Program interface

The tested program and interface is shown in Figure 4.2. The middle of the screen is a map on which the radar data is shown as objects. The different colours and symbols on the objects represent different identities (such as friendly or hostile) and types (such as aircraft or helicopters). Both the type and the identity of an object may be changed by selecting the object and clicking on the associated context menu. Each object also has information such as altitude, speed and an arrow showing the direction the object is moving in. The GUI can display this information next to the objects on the map, although this is not shown in Figure 4.2. To the right in the figure, there are some tables with information about the objects and events that have happened.

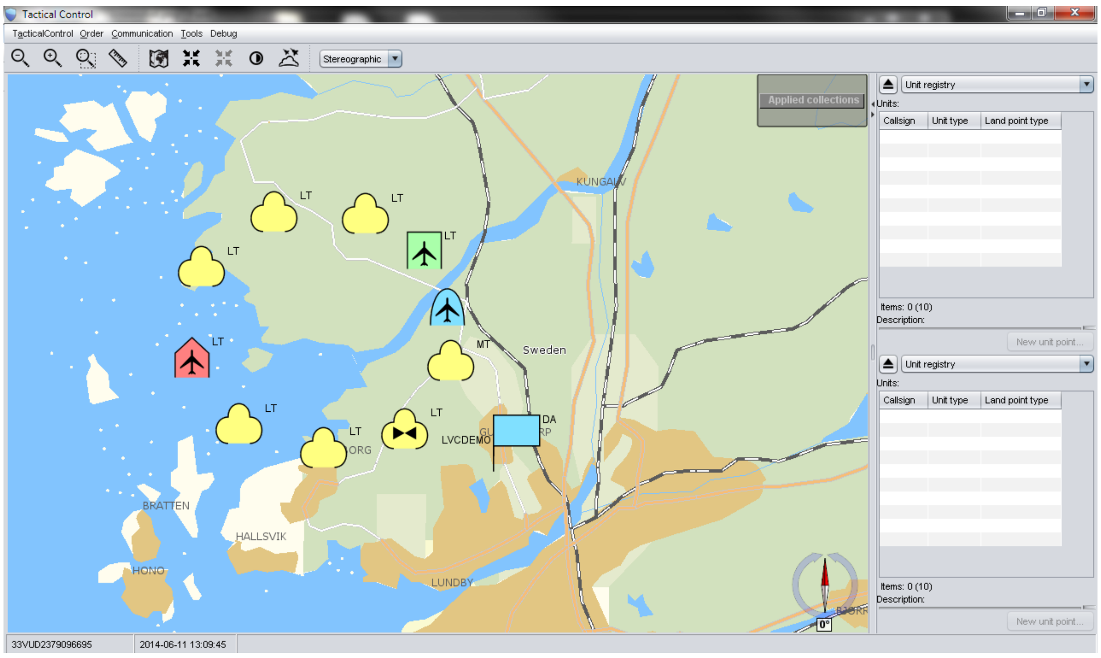
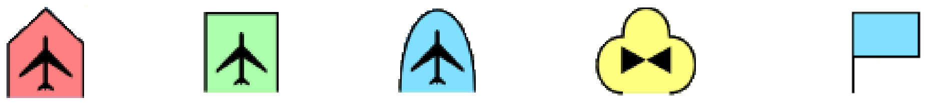


Figure 4.2: The tested program interface, running a simulation with a circle of objects, continuously moving clockwise. The tables to the right have had the data removed due to security reasons. The figure is used with courtesy of Saab AB.

The different objects on the screen are also presented in Figure 4.3 for easier understanding of the interface. Three aircraft can be seen on the map; one hostile (Figure 4.3a), one neutral (Figure 4.3b) and one friendly (Figure 4.3c). One helicopter of unknown classification can be seen (Figure 4.3d). The other objects in the circle are unknown objects of unknown classification. The flag (Figure 4.3e) to the bottom right of the circle denotes the simulated position of the radar unit.



(a) A hostile aircraft **(b)** A neutral aircraft **(c)** A friendly aircraft **(d)** An unknown helicopter **(e)** A radar unit

Figure 4.3: Explanations of the NATO APP-6A symbols in Figure 4.2.

5

Visual GUI Testing Tools

There are several different visual GUI testing tools on the market, both proprietary tools and tools under open source licenses. The aim of this thesis was, among other things, to compare a couple of these in order to determine a suitable tool for Saab AB and their GUI tests.

As stated in the introduction, the scope of this work was not to implement our own visual GUI tool since it is too complex and not suitable for this thesis. Thus, two existing visual GUI testing tools were chosen to be compared to each other: Sikuli and JAutomate. They were chosen as we wanted to test proprietary and open source tools, and they are applicable of the operating system of both the system running the tests and the tested system. These tools were also proposed by Saab AB in their Master's thesis proposal and they were evaluated in a similar but more extensive work by Alégroth [2], where Sikuli was evaluated for another subdivision at Saab AB.

5.1 Evaluation method

The comparison of the two tools was both qualitative and quantitative. The process consisted of comparing the properties and performance of each tool and evaluating them. The evaluation was performed by implementing some of the existing regression tests for the system in each of the two tools. Then the tools were evaluated with respect to empirically measured performance and perceived usability based on the tools' specifications.

The system to be tested is rather complex since it is animated. Thus, the tools' ability to handle the movement was an important part of the evaluation. The evaluation also concerns the tools' fault tolerance, and to some extent their costs in terms of development time and the learning curve for the tool.

The evaluation of the properties of each tool was done according to the model shown in Table 5.1, which closely follows the methodology and steps as done for the static tool properties in Alégroth's work [36].

Script language syntax: Which programming language is used for the testing scripts?
Cost per license per year: How much does the tool cost?
Operating system: testing computer Which operating systems can be used on the system running the tests?
Operating system: tested system Which operating systems can be used on the tested system?
Record and Replay: Does it have a Record and Replay function?
Test suite support: Does it support test suites?
Remote connection support: Does it have built-in remote connection support (VNC)?
Image representation in IDE: Are images represented by strings or images in the script?
Logging: Does it have a built-in logging function? How does it work? What is logged?
Reports: What kind of reports of the result can be created?
Documentation: How well is the tool documented and supported?

Table 5.1: Tool properties to be evaluated. Similar to Alégroth's work [36].

5.2 Sikuli

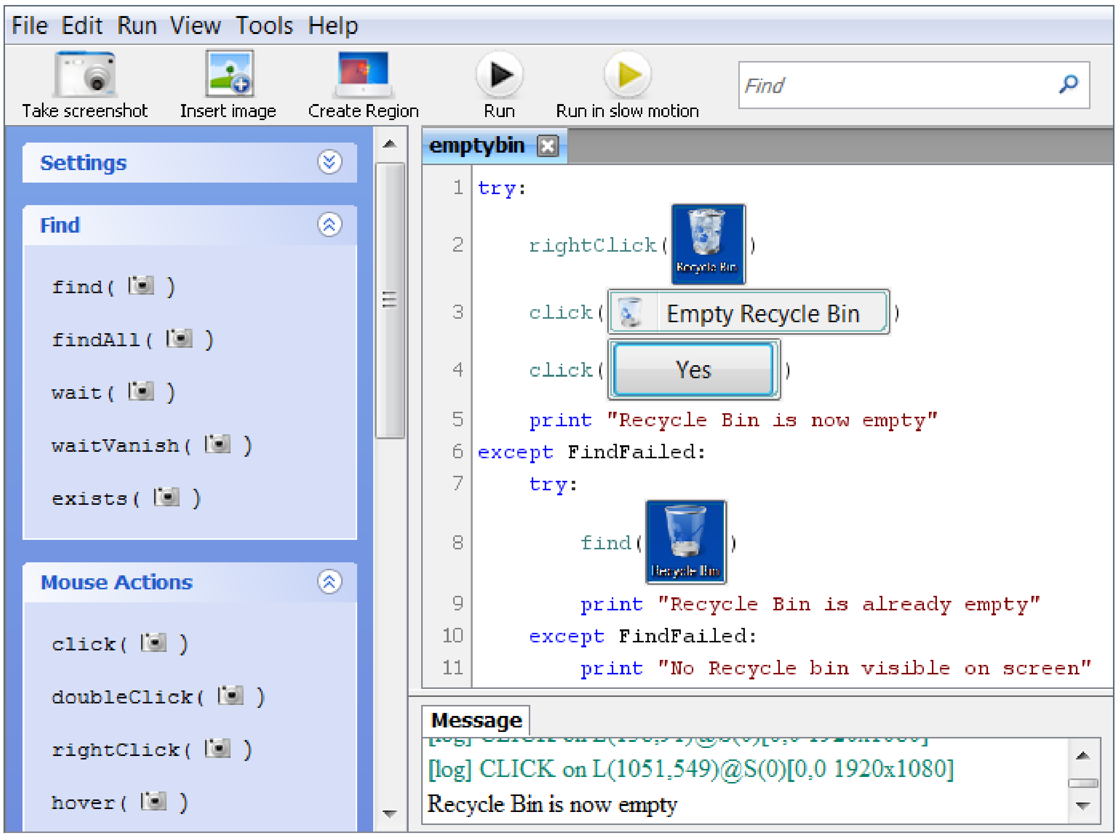
Sikuli is a free and open source testing tool developed at the Massachusetts Institute of Technology (MIT) in 2009, but is now maintained and further developed by Raimund Hocke and the surrounding open source community [37]. The project is available on the web-based hosting service GitHub. The tool is written in Jython, which is an implementation of Python written in Java [37, 38]. Since Sikuli is open source, the source code can be modified to fit the requested needs. Some properties of Sikuli, as per to Table 5.1, is shown in Table 5.2.

Sikuli uses fuzzy image recognition, which makes it possible to find a matching object on the screen even though it looks a bit different from the target [38]. The tool uses the external library OpenCV (Open Source Computer Vision) for the image recognition parts, and Tesseract for text recognition [37, 38].

Since Sikuli is written in Java (Jython), the scripts can be used in other Java programs as a library [37]. Thus it is possible to run the scripts through Java as well, with some small differences in the script code. Sikuli also integrates JUnit and thus support unit testing for GUI's [37]. JUnit tests are also used for some other automated tests at Saab AB. The JUnit test scripts are written in Python and if the script is written through the IDE, some methods for this can be generated automatically to minimise the coding [37].

Script language syntax:	Python
Cost per license per year:	Free
Operating system: testing computer	Any platform with Java support
Operating system: tested system	Any
Record and Replay:	No
Test suite support:	Yes, using import functions in Python
Remote connection support:	Not yet (in development), but can be solved using a third party VNC tool
Image representation in IDE:	Images/Strings
Logging:	No, but can be solved by writing an own logging module
Reports:	No, but can be solved by letting Python write to a file
Documentation:	Online API, Tutorials, Support

Table 5.2: Properties of Sikuli.



```
try:
    rightClick(Recycle Bin)
    click(Empty Recycle Bin)
    click(Yes)
    print "Recycle Bin is now empty"
except FindFailed:
    try:
        find(Recycle Bin)
        print "Recycle Bin is already empty"
    except FindFailed:
        print "No Recycle bin visible on screen"
```

```
[log] CLICK on L(1051,549)@S(0)[0,0 1920x1080]
Recycle Bin is now empty
```

Figure 5.1: Screenshot of the Sikuli IDE and an example script.

The IDE is shown in Figure 5.1, which also shows an example of what a script looks like. The scripting language is Python, and images can be represented as either image objects, as shown in the figure, or by their file names as strings. The IDE provides a text editor with an execution log and some shortcuts to the most common functions. It should be noted that the IDE has some known minor bugs as it is still under development.

The example script in Figure 5.1 simply empties the recycle bin in Windows. The script tries to right-click on the recycle bin, clicks on "Empty Recycle Bin" and then "Yes", and prints a success message. If it fails to find any of the images to click on, it first checks if the recycle bin is empty and prints a message describing what went wrong.

5.3 JAutomate

JAutomate is a proprietary testing tool developed by the Swedish company Swifting AB in collaboration with Inceptive AB in 2011. The tool is written in Java, and it is possible to extend the tool by adding custom Java classes [39]. JAutomate is designed for a tester, which makes it easy to learn and understand; no prerequisite programming skills are needed [40]. Some properties, as per Table 5.1, are shown in Table 5.3.

JAutomate can automate the keyboard and mouse to interact with the monitor, similar to a human user. Since JAutomate has a Record and Replay functionality, a test script can simply be created by starting the recorder and performing the test [36]. When the test is finished, JAutomate generates an automatic test script from the recording [40]. Test suites can be created by calling other scripts from a script. To make scripts reusable, parameters can be sent to a script as well [39].

JAutomate includes some features with usability in mind, such as searching for images that are not visible on the screen, e.g. by scrolling down until it finds what it is looking for. The test scripts can be exported and run through FitNesse, which is another testing tool used for automated testing at Saab AB.

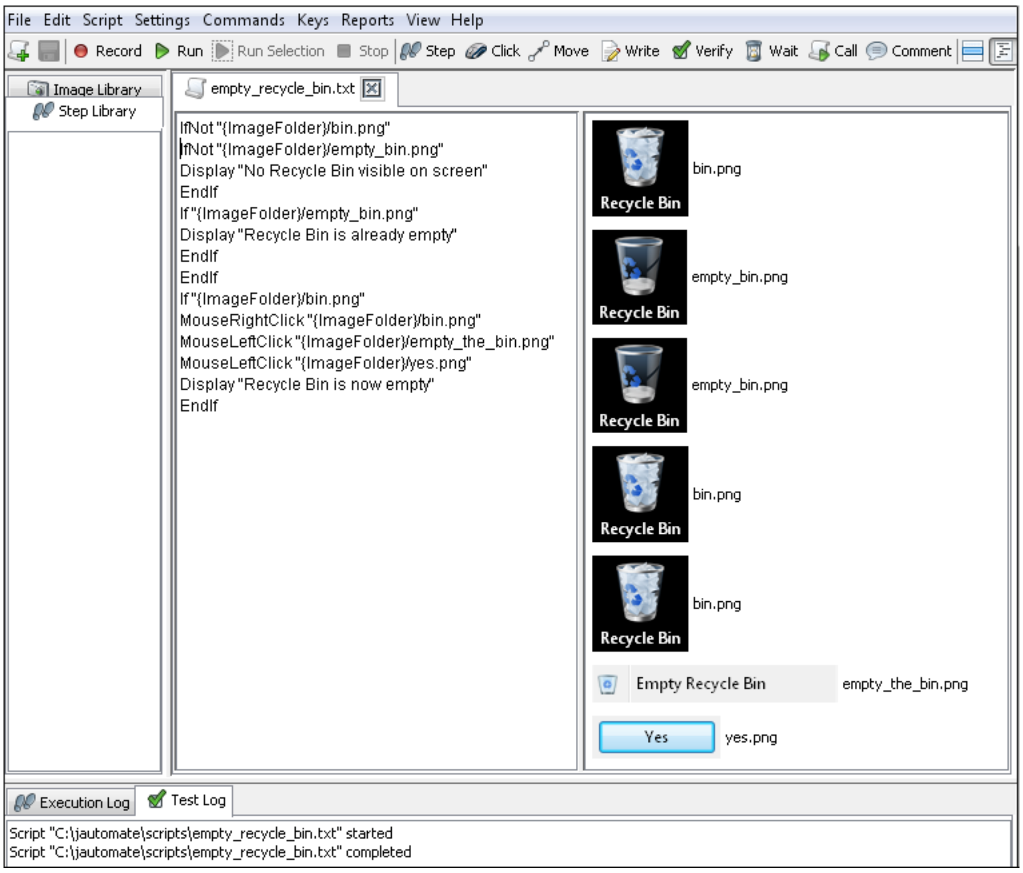
JAutomate has support for including manual and semi-automatic test steps into a script. This means that a test script can be written to execute some commands automatically, but if something unexpected happens the tester will be asked to perform the step manually and then decide whether the script should continue or not [39].

The IDE in JAutomate is shown in Figure 5.2 and Figure 5.3, which also shows an example of what a script looks like. The IDE provides an execution log at the bottom of the IDE. JAutomate has two different views in the IDE, an image view and a text view. The latter is illustrated in Figure 5.2, which shows a script presented in its text form. In this view the user is allowed to write directly in the script, like any text editor, and images are represented by their file name. However, the text view lacks indentation, colour coding and row numbers.

The same script as shown in Figure 5.2 is presented in the image view in Figure 5.3. Each row represent a command where the images to interact with is included. It is not possible to type in the image view. New commands are added from the menu of the program, or by pasting previously copied commands. The commands can be edited by clicking on them, and then changing parameters in a pop-up window.

Script language syntax:	Custom
Cost per license per year:	$995 (one specific computer) or $2495 (any one computer in the network)
Operating system: testing computer	Any platform with Java support
Operating system: tested system	Any
Record and Replay:	Yes
Test suite support:	Yes
Remote connection support:	No, but can be solved using a third party VNC tool
Image representation in IDE:	Images/Strings
Logging:	Yes, built-in
Reports:	Yes, built-in
Documentation:	API, Tutorials, Support

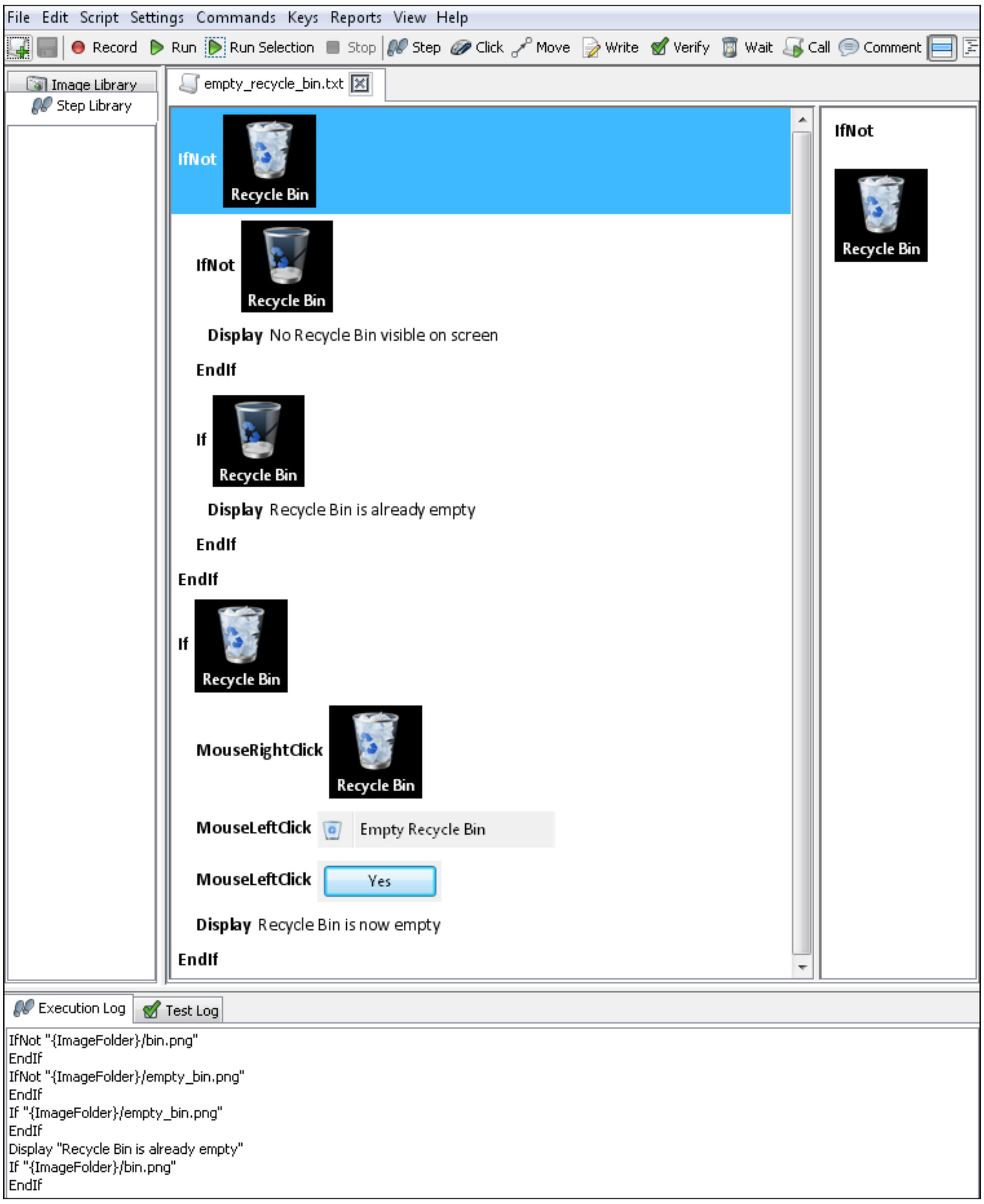
Table 5.3: Properties of JAutomate.



```
IfNot "{ImageFolder}/bin.png"
IfNot "{ImageFolder}/empty_bin.png"
Display "No Recycle Bin visible on screen"
EndIf
If "{ImageFolder}/empty_bin.png"
Display "Recycle Bin is already empty"
EndIf
EndIf
If "{ImageFolder}/bin.png"
MouseRightClick "{ImageFolder}/bin.png"
MouseLeftClick "{ImageFolder}/empty_the_bin.png"
MouseLeftClick "{ImageFolder}/yes.png"
Display "Recycle Bin is now empty"
EndIf
```

```
Script "C:\jautomate\scripts\empty_recycle_bin.txt" started
Script "C:\jautomate\scripts\empty_recycle_bin.txt" completed
```

Figure 5.2: Screenshot of the JAutomate IDE and an example script, shown in text view.


File Edit Script Settings Commands Keys Reports View Help
Record Run Run Selection Stop Step Click Move Write Verify Wait Call Comment
Image Library
Step Library
empty_recycle_bin.txt
Execution Log
Test Log

```
IfNot [Recycle Bin]
    IfNot [Recycle Bin]
        Display No Recycle Bin visible on screen
    EndIf
    If [Recycle Bin]
        Display Recycle Bin is already empty
    EndIf
EndIf
If [Recycle Bin]
    MouseRightClick [Recycle Bin]
    MouseLeftClick [Empty Recycle Bin]
    MouseLeftClick [Yes]
    Display Recycle Bin is now empty
EndIf
```

```
IfNot "{ImageFolder}/bin.png"
EndIf
IfNot "{ImageFolder}/empty_bin.png"
EndIf
If "{ImageFolder}/empty_bin.png"
EndIf
Display "Recycle Bin is already empty"
If "{ImageFolder}/bin.png"
EndIf
```

Figure 5.3: The same JAutomate script shown in image view.

The script in Figure 5.2 and Figure 5.3 performs the same action as the script described for Sikuli in Section 5.2. The script empties the recycle bin in Windows by right-clicking on the recycle bin, clicking on “Empty Recycle Bin” and then “Yes”. If it succeeds a message is displayed and if it fails to find any of the images to click on, it first checks if the recycle bin is empty and then display a message describing what failed.

5.4 Tool comparison

Based on the data presented in Sections 5.2 – 5.3, the tools were evaluated and compared against each other. The aim was to determine which tool was most appropriate to use for automating the GUI tests at Saab AB.

5.4.1 Properties

For an overview of the previous chapters and easier reading, in Table 5.4 follows a combination of the properties of the compared tools. The table is similar to the results presented in Alégroth's work [36].

	Sikuli	**JAutomate**
Script language syntax:	Python	Custom
Cost per license per year:	Free	$995 (one specific computer) or $2495 (any one computer in the network)
Operating system: testing computer	Any platform with Java support	Any platform with Java support
Operating system: tested system	Any	Any
Record and Replay:	No	Yes
Test suite support:	Yes, by using import functions in Python	Yes
Remote connection support:	In early development. Stable solution can be solved using a third party VNC tool	No, but can be solved using a third party VNC tool
Image representation in IDE:	Images/Strings	Images/Strings
Logging:	No, but can be solved by writing a custom logging module	Yes, built-in
Reports:	No, but can be solved by letting Python write to a file	Yes, built-in
Documentation:	Online API, Tutorials, Support	API, Tutorials, Support

Table 5.4: Combination of Tables 5.2 and 5.3. Similar to Alégroth's work [36].

Both of the evaluated tools work with the most common operating systems. As noted in Sections 5.2 and 5.3, both the tools can view the scripts as plain text as well as with images in the IDE.

Sikuli is a tool free of charge, while JAutomate could be expensive if many licenses are needed. An informal discussion of the cost/benefit analysis follows in Section 8.2.

Saab AB has expressed that the ability to modify the tools to suit the future needs is a good advantage. This can be done with Sikuli since it is an open source tool. JAutomate can be made more flexible to lesser extent by adding new Java classes, where the desired functionality has been implemented in Java code. JAutomate has a special package and directory where new classes to be added are to be placed. These classes are then automatically added to the tool and included in the IDE, and can be used like any other JAutomate function.

JAutomate has a Record and Replay mode, a functionality that Sikuli lacks. The Record and Replay mode records all interactions that the user makes with the system, and turns these into an auto-generated test script. Thus, rather than writing test script code, the script developer can simply interact with the system as a tester manually would, and JAutomate will generate the script code automatically. The functionality was found to work quite well and is an effective way to construct a script. However, it makes some mistakes, such as missing or misinterpreting instructions if the user is too fast, and it generally does not take mouse hovering into account, even though that is a valid input to the system, e.g. when opening sub-menus. However, the Record and Replay interface clearly notifies the user that the script needs to be manually verified before it can be used. We perceive the number of mistakes in the Record and Replay functionality to be few in relation to the gain of auto-generating parts or the whole script. We have chosen not to evaluate this functionality extensively as the generated code will not be fault tolerant enough to be suitable for a system as complex as the one at Saab AB, and thus not very applicable to the tested system. We perceive Record and Replay to be more of a convenience for the user, than a necessary feature.

A few GUI testing tools with built-in remote connection support exists. The stable version of Sikuli as well as JAutomate both lacks it, but it is in development in Sikuli. However, according to the project, it is *"very experimental, not yet complete and not fully tested nor documented"* [41].

Users often want to run the test cases in form of test suites. This allows users to create a clear and logic distinction between different tests, and allows moving parts of the test code to different logical units, which among other things improves readability of the script. Test suites also allow the users to set up mutual conditions for multiple tests. JAutomate supports tests suites out of the box, which Sikuli does not. However, a user can easily import other scripts in Sikuli directly from the Python language, so the same effect can be obtained from Sikuli as well.

Sikuli uses Python as the test script language, which makes the tool suitable for a programmer. Since it is an actual programming language, anything can be implemented, thus making the development possibilities huge. Sikuli uses images in the IDE, which makes the scripts easier to read and more intuitive. The IDE is still under development and some minor functionality does not work as expected. Some examples of this is that navigating the text using the arrow keys might place the cursor on unexpected places, some dialogue boxes are slow and some parts of the IDE acts in non-standard ways. While this causes some frustration, it has not been a problem to work with. JAutomate uses a custom test script language that is intuitive even for those that are not familiar with programming. JAutomate supports two different views in the IDE; one view that is beginner-friendly where images are shown and where new commands are created via the menus in the GUI, as well as a text view, where images are shown by their file names and where new commands are typed with the keyboard. The image view is very simple to read, but not very effective to write the scripts in. The text view is easier to use when constructing the scripts, but due to lack of indentation and code colouring the view is not very readable. The script files for both JAutomate and Sikuli are text files, so a user who is not comfortable in the respective IDE's has the option to use his or her favourite text editor to edit or create a script, and then load it into the programs.

JAutomate has a built-in automatic logging functionality, and is able to generate a number of different result reports. It will non-intrusively automatically document all steps in a script, and take a screenshot of the visible screen when failing to find what it was searching for. The user also has the ability to log anything they want by using the `Log` command. The data is written to a CSV file, which is a structured plain-text format for storing tabular data. JAutomate thus combines a nonintrusive automatic background script logging with an option to let the script developer explicitly output anything to a file. The automatic background logging is useful for debugging, however it is less suitable for running longer and bigger test scripts. The `Log` command can be used for this, but this command does not include screenshots which in our complex environment is very important in order to determine the cause of errors, if they occur. This can be solved in both tools by calling an external recorder program. Sikuli lacks a logging and result functionality all together, but the user has the ability to implement this in a Python module. However, it does not have an automatic logging feature, and the user must call the logging function every time something is to be logged. To manually call the logging function when logging is needed has both advantages and disadvantages. The advantage is that the user may decide what to log and when, anywhere in the script. This makes it possible to have the logging tailored to the users' needs. The disadvantage is that the user must call this logging function in the script every time it should be used, which may result in code cluttering. However, this was not a big disadvantage at Saab AB, since the number of occasions where the logging was needed was limited. By carefully considering what to log and when, the number of log events could be constrained. The logger we implemented is described further in Section 6.

The OCR functionality for both the tools was also tested. In Sikuli it is still experimental, but the OCR engine Tesseract is used, which is considered as one of the most efficient open source OCR engines available [42]. The results for both of the tools were poor, giving inconsistent results for the same input and sometimes failing all together.

Both Sikuli and JAutomate has an API documentation and tutorials, where Sikuli's are stored online and JAutomate's API documentation is bundled with the software, with some tutorials available online. JAutomate is supported through its owning company. Sikuli is supported through a forum where both the developers and other users can answer questions. The Sikuli API documentation is more detailed than the JAutomate API documentation, and we were able to find solutions to problems faster for Sikuli than JAutomate. However, both of the tools were considered good in this respect.

5.4.2 Performance

The performance, with respect to robustness, for the tools was measured for two different test suites of different size and complexity, which are representative for a large part of the regression test verification specification. The two test suites contain 49 test cases and 6 test cases respectively. Each of the test cases perform different actions and check for different responses. An example of a test case would be to carry out an assignment that is denoted as risky in the system, and make sure that the remote operator correctly receives a warning regarding the assignment. The test case then handles the warning appropriately. A different test case would test other functionality, thus the test cases focus on different parts in the system.

All the test suites were written in an informal language in the regression test verification specification. Therefore, they were translated into a more formal language, i.e. implemented. Due to this translation, the implemented tests are not completely identical in the two tools, and differ somewhat from manually conducting the tests. The implementations of the tests and the test verification specifications tests the same functionalities – but in slightly different ways. The reason for the differentiation is because of the high complexity of the tested system, which requires the implementations to be adapted both for the system and the VGT tool. The implemented tests are further described in Chapter 6.

Both of the evaluated tools and the tested system were optimised to increase the accuracy and performance. The tested system had the map background disabled (i.e. the background was of uniform colour instead of a map) in order to avoid any confusion between the objects and the background, which increased the performance. This should not be seen as a limitation of the applicability of the VGT technique for this system. The tools works with the map background enabled as well, but to get a fair evaluation of the tools with minimal external interference and maximal performance, the map background was disabled. As mentioned in Section 4.1, the tested system by default equipped objects with some visual information, such as an arrow indicating the object's current direction. The option to show this information was disabled, improving the image recognition accuracy, as the objects became easier to find. Furthermore, mouse cursor movement was disabled, i.e. the tools placed the mouse cursor at the desired location rather than moving it across the screen. This led to a performance boost, and significantly decreased the frequency of how often the tool missed to click on a moving object by clicking beside it. Image recognition similarity parameters were optimised, so that the VGT tool performed correctly and distinguished between the different kinds of objects. In JAutomate, the parameter that made a test suite stop running in case of errors was disabled, which was especially useful when encountering image recognition failures.

The performance results from Sikuli and JAutomate, presented in Table 5.6 and 5.7 were also compared against the time taken for performing the same manual test, presented in Table 5.5. Note that the results from Sikuli and JAutomate consider 1000 or 100 iterations each, depending on the test suite, while the manual is only one iteration.

Manual test result

	Test 1	**Test 2**
Iterations:	1	1
Time:	7min 41s	50s

Table 5.5: Results from manual tests.

It should be noted that Table 5.5 only shows the time it took to interact with the system and perform the actions. The normal procedure when manually conducting tests is to take some time reading and understanding the test suites; possibly do some prerequisite preparation and continuously read the test verification specification while performing the test. Thus, manually conducting a test in practise is expected to take much longer than what is presented in Table 5.5. A more realistic time of running a test in practise has not been taken into consideration in the report for a few reasons; the amount of time spent reading the test verification specifications is dependent on the experience of the tester, how fast he or she can read and understand a text, individual traits, etc. Above all, the VGT technique is not considered due to its speed, so the extra time that a human would take was not deemed relevant for our purposes.

Sikuli test result

	Test 1	**Test 2**
Iterations:	100	1000
Time:	1d 21h 50min 25s	16h 48min 23s
#Tests passed:	98	999
#Tests failed:	2	1
#Image recognition failures:	17	7
#Misses the moving object:	82	13
#Calls to image recognition algorithm (per iteration):	~1200	66
#Calls to click function for moving objects (per iteration):	~200	12

Table 5.6: Results for Sikuli.

Test 1: The duration for the test was 1 day, 21 hours, 50 minutes and 25 seconds, for 100 iterations. Thus, one iteration was performed on average in 27 minutes and 45 seconds. Of the iterations, 98 passed and 2 failed, which gives a script accuracy of 98.0%. One failure was because there was an object that had not been assigned when it should have. This was probably an image recognition failure, i.e. Sikuli missed to discover the object, which could happen if the object is selected, thus looking different from what the Sikuli script expected. It could also be due to some error in the tested interface that resulted in Sikuli correctly assigning the object without the interface acknowledging it. There is a third possibility as well, that the delays were too small and the system did not have enough time to react on the change before Sikuli verified it. However, due to extensive logging during the run and making Sikuli perform an extra control of the object later, we can determine that this was not the case. The other failure was because the system lost the connection, which resulted in that the interfaces were no longer fed with radar data and all the objects disappeared. Sikuli failed to click on the object and clicked next to it 82 times, which is roughly one time per iteration. The image recognition algorithm failed 17 times, which gives an average of about once every fifth iteration. Neither of these should however be considered as big problems, since they are taken care of by extensive failure mitigation. It should be noted that this test is very big, explaining the long duration and the few iterations performed. Every iteration contains about 1200 calls to the image recognition algorithm and 200 calls to the click function for interacting with moving objects. Thus, missing the object around one time per iteration in average is about 0.41% of the calls made, and the failing image recognition calls are around 0.01%.

Test 2: The test took 16 hours, 48 minutes and 23 seconds to perform 1000 times. Thus, one iteration took about 1 minute to perform. Of the iterations, 999 passed and 1 failed which gives an accuracy of the script of 99.9%. The failing iteration appeared when an assignment failed, and thus the sought object could not be found. The reason for the assignment failing is unknown and could depend on either the tested system not registrating the performed action or Sikuli not performing the action, for instance due to image recognition failure (i.e. the object was not found). Had we used a third-party recording program during the testing, we could have determined the cause of the failure. In order to continue the test scripts upon a failure, the system must be reset to a known state whereupon a new iteration can be started. If that fails, it is convenient to exit Sikuli. All of the seven image recognition failures appeared when Sikuli did not find all of the objects, even though they existed. However, this was solved in the code by trying to find the objects several times which in all cases succeeded on the second try. Sikuli also missed to click on the object at times, instead clicking beside it, which resulted in a different menu appearing. This happened 13 times during the 1000 iterations. Every iteration contained 12 click operations for interacting with moving objects which gives an average of missing the object about 0.11% of the calls made. Similarly, seven image recognition failures during 1000 iterations with 66 calls to the image recognition algorithm per iteration gives in average a failure of 0.01% of the calls. All of these failures could be solved by reattempting the action. This statistic should be noted when making new scripts, as this kind of fault happened from time to time and thus must be resolved.

JAutomate test result

	Test 1	**Test 2**
Iterations:	100	1000
Time:	2d 16h 9min 42s	1d 15h 52min 22s
#Tests passed:	95	994
#Tests failed:	5	6
#Image recognition failures:	0	5
#Misses the moving object:	674	228
#Calls to image recognition algorithm (per iteration):	~2000	72
#Calls to click function for moving objects (per iteration):	~300	12

Table 5.7: Results for JAutomate.

Test 1: The test took 2 days, 16 hours, 9 minutes and 42 seconds to perform, for 100 iterations. Thus, one iteration was performed on average in 38 minutes and 30 seconds. Of the iterations, 95 passed while 5 failed, giving an accuracy of the script of 95.0%. Two of the failures were due to bugs found in the system. One led to a menu ceasing to work, and one was a glitch in the connection. The latter resulted in that an object got duplicated, in a way that was not supposed to be possible. The object did not answer to commands, and as opposed to the other objects, it did not move. Three of the failures occurred when an assignment was supposed to result in a conflict being detected, but JAutomate failed to see the result. The reason for this is unknown, but the most probable cause is image recognition failure, resulting in that another object may have been mistaken for the sought image. It is also possible that an error in the tested system was found. Had there been screenshots or a recording of the script while running, analysis could have revealed the cause. Apart from these three mentioned cases, the image recognition algorithm never failed to find the sought object. The number of calls to the image recognition function during the script is many: about 2000 calls per iteration. Thus, we suspect there is reason to doubt that number. Due to the limitations of the logging function, and the difficulty of writing fault tolerant scripts in JAutomate, we consider it a possibility that image recognition failures occurred that were not detected by the script. JAutomate missed to click on the moving object 674 times, which gives an average of about 6-7 times per iteration. However, though they are many it is not considered a problem since this is handled by the failure mitigation in the script. It should also be noted that this test is extensive which explains the long duration and why fewer iterations were performed for test 1 than for test 2. Every iteration contains about 300 calls to the click function. Thus, missing the object about 6-7 times per iteration in average is around 2.2% of the calls made.

Test 2: The duration for the test was 1 day, 15 hours, 52 minutes and 22 seconds, for 1000 iterations. Thus, one iteration was performed on average in 2 minutes and 24 seconds. Of the iterations, 996 passed while 6 failed, which gives an accuracy of the script of 99.6%. Four of the failures happened because of different kinds of image recognition failures. Two of them were because the sought object was selected, making it look different than what the JAutomate script expected. The other two were hard to determine due to a lack of screenshots or recordings of the script running. However, the most probable cause is either that the image recognition failed, or that the JAutomate script attempted to verify the action before the system had finished performing it. One of the two other failures was because the system lost connection, which resulted in that the system ceased to be fed with radar data, whereupon all objects disappeared. The last failure is also hard to determine the cause of, due to the lack of screenshots or recordings. The failure appeared during an assignment of an object from one identity to another, which was supposed to result in a conflict but for some reason the conflict was not detected. The reason why JAutomate failed to detect this is probably due to image recognition failure, but it cannot be precluded that it was not due to a bug in the system. The image recognition algorithm failed to find the sought object 5 times, even though the object existed and was visible. Every iteration contains 72 calls to the image recognition algorithm, which in average gives a failure of around 0.007% of the calls made. This is not a problem since it happens rarely and it is handled by retrying the action, which succeeded in all cases. JAutomate missed to click on the moving object 228 times, which gives an average of about once every forth iteration. This is also handled by retrying the action, which eventually solved all such cases. Each iteration, 12 calls to the click function are made for interacting with moving objects. Thus, missing the object about once every forth iteration in average is around 1.9% of the calls made.

Comparison

For an overview of the measured performance and easier reading, Tables 5.8 and 5.9 contain a combination of the measured performance of the compared tools when running the two test suites.

The accuracy is very similar both between the different tests performed and between the tools, which can be seen in the results presented in Table 5.8 and Table 5.9. However, the time taken for running a test in JAutomate is about twice as long as for Sikuli. The speed of the image recognition algorithms depends on the CPU, and the difference in speed between Sikuli and JAutomate runs may be due to algorithms being implemented in different languages. The image recognition algorithm in JAutomate is implemented in Java while the image recognition algorithm in Sikuli is implemented in C++, which is faster. However, as mentioned in Section 5.4.2, there are parameters and settings that can be tweaked in the tools, which affect the performance. We have tried to optimise these to be as similar as possible. We had very few image recognition failures using JAutomate in relation to Sikuli. However, we cannot guarantee that all such errors are found in our scripts due to the limitations in script expressivity that makes it hard to catch all failures and due to the limited number of screenshots taken during a script

Test 1	**Sikuli**	**JAutomate**
Iterations:	100	100
Time:	1d 21h 50min 25s	2d 16h 9min 42s
#Tests passed:	98	95
#Tests failed:	2	5
#Image recognition failures:	17	0
#Misses the moving object:	82	674
#Calls to image recognition algorithm (per iteration):	∼1200	∼2000
#Calls to click function for moving objects (per iteration):	∼200	∼300

Table 5.8: Results for Test suite 1.

Test 2	**Sikuli**	**JAutomate**
Iterations:	1000	1000
Time:	16h 48min 23s	1d 15h 52min 22s
#Tests passed:	999	994
#Tests failed:	1	6
#Image recognition failures:	7	5
#Misses the moving object:	13	228
#Calls to image recognition algorithm (per iteration):	66	72
#Calls to click function for moving objects (per iteration):	12	12

Table 5.9: Results for Test suite 2.

execution. Also, there is a small difference in the tested systems that may explain the difference between the tools as well. When running the Sikuli tests, an option to enable track history is activated which resulted in a small "marker" sometimes appearing on the objects. This may have affected the results, but since the number of image recognition failures are few we did not consider this a problem when running the Sikuli tests. However, with this option enabled when running the JAutomate scripts, a very large amount of image recognition failures occur. Thus, this feature is turned off for the JAutomate tests. This must be taken into consideration when comparing the results. For visualisation of the issue, Figure 5.4 shows an example of an object with the "marker" (Figure 5.4a) and without the "marker" (Figure 5.4b).

(a) A friendly aircraft with "marker" **(b)** A friendly aircraft without "marker"

Figure 5.4: Visualisation of the difference of an image with track history enabled and with track history disabled.

Both of the tools make interaction mistakes from time to time, which are expected since the tested system contains moving objects. For test 1 and 2, JAutomate missed to click on the moving objects around 8 respectively 17 times more often than Sikuli. We anticipate that this is due to the differences in the image recognition algorithms, which also explains the longer duration for JAutomate since an iteration takes longer time if mistakes are made. There are also implementation differences that could have impact on the time taken for an action. Due to the scripting language in JAutomate, achieving fault tolerance in the scripts is more complicated. To exemplify; if Sikuli fails to click on a sought image, it will throw an exception that the coder can catch. In JAutomate, one needs to follow the click action with a check to see if the action happened as expected. The Sikuli approach means that the code "notifies" of failure, while JAutomate needs to perform the slightly costly operation of checking for the correct result after each attempt at an action. Thus, the JAutomate implementation is less effective, which can explain the difference in duration.

5.4.3 Fault tolerance

During script development, we found that errors and faults would often arise in script development. Thus, for a VGT tool to be usable, it needs to be able to handle the different kinds of faults that can appear. How well the tools can handle this largely determines how applicable they are. As mentioned in Section 3.4, the scripts were constructed as a series of states, which are executed in order. In any of the states, errors might appear.

When evaluating the tools, the following sources of faults were identified:

Fault type 1 Image recognition failure.

Fault type 2 The script behaves slower/faster or less intelligent than a human tester.

Fault type 3 Unexpected system behaviour.

Fault type 4 External interference.

Fault type 5 Undeterminal state.

A short discussion of the different kinds of faults follows on the next pages.

Fault type 1: Image recognition failure can occur on any given point in the script. Most of the time the image recognition algorithm fails to find the sought image, despite it being visible (False Negative in Figure 3.4 in Section 3.4). Less commonly in our tests, but more dangerous, is when the algorithm finds the wrong image (False Positive in Figure 3.4). The script developer can set image recognition parameters to control how sensitive the algorithm is. By tuning these parameters, the script developer may overcome image recognition faults. We found that one static value of a parameter often did not work through the whole script, so in certain places of a script it had to be changed to be more or less strict. However, the process is unintuitive and frustrating; while the task of changing a parameter is easy, it is not obvious to the script developer why the old value would cease to work in a seemingly arbitrary point in the script. Due to the moving objects, the image recognition may fail after the script has been running for some time, so it is a lengthy process for the script developer to tune the parameters. This is in contrast to traditional software testing, e.g. unit tests, where the developer often immediately can determine whether the test works or not.

More difficult to detect is when the image recognition algorithm finds the wrong image. Since the script thinks that it has found what it was looking for, it may result in unexpected results later. Depending on the tested system and the current test, the whole test script may fail. If the script tries to interact with the object it was trying to find, the interaction may fail or produce unexpected results. The script developer can be wary of the possibility of this scenario, and create code to check for the wrong events happening, however the checks may be hard to implement and the checks themselves can fail. If the error is detected, any actions taken on the system may need to be reverted, and the code can retry performing its action. If the image recognition is done in order to verify a result rather than acting upon an object, the implications of a failure may be more serious, depending on the tested system. If the script incorrectly thinks it got the expected result, the test may be reported as successful, when it should in fact have failed. It is very difficult to predict this, and the test scripts needs to be of a quantitative nature, i.e. repeating the tests multiple times.

Fault type 2: The script might behave either slower or faster, or less intelligent than a human would in multiple cases. For instance, when running tests on the system at Saab AB, some tests would find moving objects and click on them in order to interact. Sometimes, the script was a bit too slow, and even though the object was found as expected, it had moved in the meantime, resulting in the click missing the object. A human tester would realise that he or she needs to click on the object, and not act as rigid as a VGT script. We anticipate that these kinds of faults become more and more of a problem, the more animated an interface is; moving objects and/or more “advanced” ways that the system can communicate with the user would pose more of a problem for a VGT script to handle. The VGT technique itself is usually quite fast, which also may be an issue if this is not taken into consideration. The tested system is often slower than the test script, and delays in the script may be needed to compensate. For instance, if the script performs an action that causes the system to visually change, it might take a little while before the system has reacted to the action. If the VGT script immediately starts looking for the visual change, it might not be present, whereupon the script fails.

Fault type 3: The tested system can generate a number of different events, such as errors, messages or notifications. If these are valid events, a test script should be able to handle them accordingly; if not, the test script has found an error in the tested system. The more complex the tested system is the more is demanded from the script developer in order to predict all valid states. The more uncommon the event, the less important it probably is to handle; if a test fails because the script is unable to handle an uncommon event, it may have little impact on the overall testing procedures. For instance, if the tested system can legally warn that the available hard disk space is running low, this scenario might be uncommon and not something that a script developer takes into consideration.

Fault type 4: External interference may come from a number of sources. This is very difficult to predict, since it is a collection of a number of different events, such as errors, messages or notifications that the operating system or other running programs might generate. It may be practically impossible to predict all different events that might occur. As with Fault type 3, it makes little sense for a script developer to spend a lot of time handling these faults, as long as they are rare.

Fault type 5: The last fault type is very hard to handle. An undeterminal state is a state of the system that cannot be determined properly. This could happen if a system does not visually change after an action has been performed. The VGT tool can only confirm visual changes on the system, not that the action itself was performed. This fault can also appear if the visualisation of a change is very short, so that the VGT tool does not have enough time to confirm the change. Several examples of this can be found in the tested system. For instance, there is a feature to discard the local data about an object. If one of the systems still detects it, it will continue to transmit the object over the network. Discarding an object that is present on the network will visually look as if it disappears for a short while, only to quickly appear again. The time is too short for the VGT tool to be able to detect the change, so it would not be possible to confirm the success of performing this action in the system.

5.4.4 Moving objects

As described in Section 4.1, the tested system consists of a map, where objects detected by the radar move across the screen. Many of the tests will interact with the objects in different ways, for instance selecting them with the mouse and right-clicking on them to bring up a context menu. An object is shown in a position for about one second, and then its location is updated; in other words, objects move in “ticks”. For the VGT tools to be applicable on these systems with moving objects, they must be able to work with animated interfaces.

Neither JAutomate nor Sikuli have any problems handling the animated tested system. Upon searching for a target image on the screen, both tools take a screenshot of the screen, and perform the image recognition algorithm on it. This provides stability and robustness for the image recognition. However, this has some consequences regarding how one can act with the system in the subsequent test scripts.

Since the image recognition algorithms were carried out on a static screenshot, the objects found may have moved during the time the algorithm was working. This means that the subsequent actions that the test script tries to perform with the object, are carried out in the location where the object was a few moments ago. Whether this causes problems, and what consequences this might have, depends on the tested system. The following section will discuss this further.

Problem mitigation

The core of the problem with moving objects – that they might have moved during the time the image recognition algorithm was performed – cannot be overcome when using image recognition based systems. However, we have identified four ways of mitigating the problem, presented in the following paragraphs.

If the script occasionally misses the object but usually succeeds in its attempt to interact with it, the coder can make the code look for interaction failures. If the object has moved and the test script missed it, the code could retry the operation, and hopefully it is slightly faster this time. Since the tested system is animated, the algorithm might perform somewhat different upon subsequent searches for images; thus a retry might be worthwhile. Due to the objects moving in "ticks" in the tested system, the object might have moved somewhat longer if the image search was begun right before or right after a movement "tick". This is emphasised by the synchronisation of the systems, where one system could fall somewhat behind the other and the object's positions would be updated in an irregular manner. The main downside of repeatedly trying to find the object when failing to interact with it is additional code complexity, and that the test script could reach a non-terminating loop of repeatedly trying to find the object, attempting to interact with it and then failing to do so due to slowness. The number of interaction failures, in percent, that we could solve in the tested system using this approach is shown in Figure 5.5. The data shown in the figure was collected during the execution of the test suites for Sikuli and JAutomate. Already after one retry attempt, 67% of the interaction failures are solved. By allowing four retry attempts, over 95% of the interaction failures are solved. However, it must also be taken into consideration that more retry attempts results in a slower script. Furthermore, a human would probably not make very many retry attempts, so too many attempts would not be reasonable if human behaviour is to be simulated. If that is the case, the number of retry attempts should be chosen carefully.

Since the movement is a problem, a way of mitigating it could be to make the movements as short as possible. In the tested system, this could be done by increasing the size of the object symbols; thus the area one can click on in order to select an object is bigger. In the tested system, the zoom level could also be decreased. The more zoomed in the system was, the greater would the distance be that an object moved in each "tick". A decreased zoom level would cause the objects to move a shorter distance across the screen. In the tested system, the zoom level had to be set with some care; zooming out too much would cause objects to be placed on top of each other, making it impossible to distinguish them from each other.

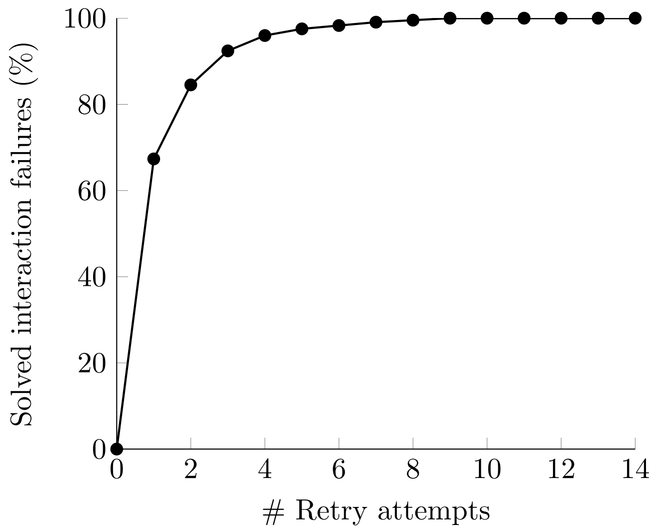


Figure 5.5: Graph showing percentage of the interaction failures can be solved by using different numbers of retry actions.

The time the algorithm takes can be improved in different ways. Some ways this can be done is by improving the hardware, terminating unneeded programs or processes in the background, reducing the area to be searched, optimising the tested system (if the user can modify its source code), or by placing the VGT tool on a different computer and running it via VNC so that the VGT tool operations doesn't burden the tested system. If the algorithm performs faster, the object is hopefully still in the location where the algorithm found it, when the subsequent interaction is to take place. The drawbacks of this, in the order given, is that improving hardware can be too expensive, there might not be any programs or processes that can be terminated, it might not be possible to reduce the area to be searched, it might be unfeasible to optimise the tested system, and running via VNC may not be possible or still not reduce the time taken enough. The time taken for using VNC could even be increased due to the amount of data that would need to be transmitted over the network.

The subsequent code can try to predict where the object might have moved to. If the code knows the direction of movement, or can make an educated guess about it, it might be able to predict where to try to interact. The drawback of this is that it might not be possible to know or guess where the object is going, and even if it can be done, the code complexity might grow substantially. In our tested system, it could be possible to implement this, but we have not had a reason to, since the mitigation method of simply retrying to interact worked good enough in all cases. However, worth mentioning is that upon failure to interact properly, when we tried to find the new location of the object, we did "predict" where to look for the new location of the object. Since our tested system was just a bit too slow, the object could not have moved too far, so rather than looking for it on the whole screen, we took the location of where it previous was and extended the region around it a bit. Thus, we could speed up the time it took to find it again by making a "prediction" that it had not moved too far. While this worked for relocating the object, it was not possible to interact with it using this technique in the tested system. Interacting had to be done on the object and not nearby.

5.4.5 Development time and other perceived costs

A crucial part when evaluating the tools was the time and cost aspects. The development time for creating a script and the extra time needed for making the script robust must not get too high. To be useful, the testing script must to some extent contain failure mitigation, or robustness, which we define as "a measure of the ability of the code to withstand the different types of faults". This of course increases the development time for implementing the test scripts, as stated in Section 5.4.3. While robustness will be paramount in order to create useful code for a complex system, spending too much time adding robustness might not be cost effective. A failing script is not expected to be a cost in an industrial context, and as stated in Section 3.4, scripts should be designed to allow for failures, despite nothing being broken in the system. While spending exaggerated time on this form of robustness might not be cost/time effective, we however note that robustness allowing for updates of the tested system without the need for script updates would be cost/time effective. The development time for creating a script is affected by the prior knowledge – both of the tool and script language as well as knowledge about the system and the test case to be created. By having knowledge about the test case and the system, the script developer will be able to predict possible outcomes and faults that could appear (for instance when working with moving objects). Then, failure mitigation could be added directly during the development process as a result of this knowledge.

The development time for our implemented test suites in the tools were about the same, for constructing the script without any failure mitigation. However, adding failure mitigation and generalising the code took significantly longer time in JAutomate than in Sikuli due to the limitations in the scripting language in JAutomate.

Another cost to consider was the learning curve of the tool, both when it comes to getting familiar with the tool and with respect to the powerfulness of the written scripts. This is of course very individual, since people have various experiences of programming, writing test cases and testing tools. Different people might also prefer different tools. We can only speak for ourselves, and we have similar background which include programming experience but neither of us had any notable experience in the concerned scripting languages. We thought that JAutomate had a faster learning curve, since it is more intuitive than Sikuli. However, we thought that Sikuli was easier to use to create powerful scripts, partly due to the complexity of the tested interface with moving objects.

5.4.6 Conclusion

Since image recognition cannot be fully trusted, tests need to be quantitative in an industrial context, i.e. run repeatedly. If the result is close to 100% success, the failing runs can be examined manually. In our case, most of the failures are due to image recognition failures. There are some failures that are hard to explain and can be due to that either the tool failed some action or that the system did. Despite some failures from time to time, the VGT technique is very promising with good results from both of the tools.

The two chosen tools that were evaluated are actually pretty similar, which was also concluded in the more extensive study presented in Alégroth's report [2]. But there are some differences, as described in the following paragraphs.

When it comes to performance for the tools, the accuracy is about the same but the number of interaction failures is higher for JAutomate than for Sikuli. The duration for the JAutomate scripts are longer than the duration for the same Sikuli scripts, which is expected since a high number of interaction failures will take a longer time. The number of image recognition failures is higher for Sikuli than for JAutomate, however we cannot determine if this is because of the algorithms or due to some differences in the tested system – for instance the tracking marker or background processes in the system. Anyway, we are satisfied with the results from both of the tools in the performance perspective and due to the small differences we consider the tools equal in this aspect.

JAutomate has a more intuitive scripting language than Sikuli, which makes it easier and faster to learn – especially for a novice programmer. The scripting language in Sikuli is much more powerful and it is easy to write advanced code in Sikuli. None of us had any notable experience with Python before, but we both thought that it was very easy to learn, and it includes many usable functions, good API and simple syntax. JAutomate, on the other hand, has its main drawback when it comes to the scripting language from a programmer's point of view. It lacks functionality needed for more advanced scripts, such as data structures like arrays and methods with return statement. Variable arithmetic is very ineffective due to the implementation of expression evaluation, which requires several commands for just assigning a value to a variable. This also affects the development time for creating scripts in the tools. For more advanced scripts using general functions and with failure mitigation, the development time in JAutomate is significantly higher than in Sikuli – which is due to the limitations in the scripting language.

JAutomate has a stable and robust IDE with two different views. The image view is well suited for a novice programmer, while text view on the other hand, is not adapted for a programmer. The view lacks indentation, colour coding, row numbers and images which makes the code difficult to read for large scripts. Sikuli also has images in the IDE and a nice interface and editor, but the IDE has some bugs since it is still under development. Both of the tools have good image recognition engines, which are shown in the performance tests in Section 5.4.2 but neither of the OCR functions work very well. JAutomate has a feature called AI images, which makes it possible to tell the image recognition algorithm to pay more attention to a specific part of an image. This results in that the matching is performed more or less on what is important in the image, rather than the whole image which often include some background which is unimportant. JAutomate has a built-in logging and report function, which is very helpful for debugging. However, the report is a bit difficult to follow for advanced scripts and cannot be adapted. Sikuli does not have a built-in logging or report function, and it compiles during runtime, which can make it hard to find script errors. However, a logging module can be implemented by the user. It is also valuable that JAutomate can export the tests to other testing frameworks and has a record and replay function, which may ease the script development. Sikuli does not have these functions built-in, but it is open source so the tool can be further developed and it has good functions and a clear API.

Since the tested system is complex, it requires the tools to be fault tolerant and to handle things like moving objects. Functionally, both of the tools handle this but due to the powerfulness in Sikuli's scripting language we are able to handle moving objects in a better way. The strength of the scripting language also affects the fault tolerance and partly the development time. Aside from this, the tools are more or less similar and which tool to use is a matter of choice, which depends on the person using it. However, due to the strength of Sikuli's scripting language and the script editor, Sikuli is chosen as the most suitable tool for the tested system.

6 Implementation

THE IMPLEMENTED CODE PARTLY CONSISTS of some test suites, implemented for both Sikuli and JAutomate. Since the open source tool Sikuli was chosen as the recommended tool to use for the tested system and due to the poor results of the Optical Character Recognition (OCR), a modified OCR functionality for Sikuli was implemented. Also a module for logging and generating reports of Sikuli scripts was included in the implementation. The implemented test cases and descriptions of the other implementations are further described on the following pages.

6.1 Test suites

With assistance from Saab AB, two test suites were chosen to be implemented from the regression test verification specification for the system. These test suites were chosen since they are some of the most extensive tests, are rather straightforward to automate and because they are representative for a large part of the regression test verification specification for the tested system. The chosen test suites are representative since they test the most complex functions in the system, i.e. the interactions with the moving objects, which is widely used in the test suites specified in the regression test verification specification. Furthermore, they test basic functionality on static components which is also widely used in the system. Thus, if those tests are possible to automate, then so is a large part of the remaining test suites – where many of them are much easier or contain identical or at least similar operations as the one performed in the implemented test suites. Furthermore, the chosen test suites consists of several test cases (49 respective 6) which explains their extensiveness. Due to the tests' size and the large cost and time this requires, these tests are seldom run today. In their nature, they are also perceived by Saab AB employees as some of the most boring ones to carry out manually.

The tests were performed using two systems sharing the same data, i.e. communicating with each other. The VGT tool was run on one computer, where one of the systems was running. This system is referred to as “local”, as it is on the same computer as the VGT tool. Via a remote desktop connection, another system’s monitor was visible on the local computer’s screen; this system is referred to as “remote”. These systems constantly shared which objects they had discovered. Each object had an identity and a type. The two systems could change these locally, as well as transmit their apprehension of the objects. Some identity changes must for security reasons be flagged as conflicts, which can only be resolved by an operator. Table 6.1 shows the identities and types that objects can have in the system. Any object can have any identity and type.

Identities	**Types**
Pending	No statement
Unknown	Electronic Warfare
Assumed friend	Missile
Friend	Fixed wing (aircraft)
Neutral	Rotary wing (helicopter)
Suspect	UAV
Hostile	

Table 6.1: The identities and types that objects can have.

Test suite 1: The first test suite was performed by changing the identities of the aircraft detected by the radars. The two systems should agree on which identity each object has. The test suite aimed to verify that if the remote identity of an object differed from the local, the correct action should be performed depending on the identities. The possible actions that could be performed was to simply accept the remote data and store it as local (i.e. change to the same as the remote data), reject the remote data (if the remote data was undefined) or conclude that a conflict existed which needed be resolved by an operator. If the remote data was the same as the local data, no action needed to be taken. A conflict exists if for instance the local system has identified an object as friendly while the remote system has identified it as hostile. There are seven different identities an object can have (see Table 6.1) and changing between all these logically results in a test matrix of $7 \cdot 7$ tests, i.e. 49 different test cases. Running the test suite verified that the proper action was taken when any identity was changed into any other.

Listings 6.1 shows pseudo-code for Test suite 1 implemented in Sikuli, where `matrix` is the mentioned matrix of identities. The rows and columns represent what identity to change from and to. The matrix values determine the action to take, where 0 denotes ’no action’, 1 denotes ’conflict’, 2 denotes ’accept’ and 3 denotes ’reject’. The `imgs` array contains images of each object identity, and `assignIdentityToAllObjects` and `assignIdentityToOneObject` are functions implemented to assign objects. Furthermore, `conflict`, `accept` and `reject` are functions for detecting and handling the expected system reaction. From `sikulilogger`, the logging function `log` is imported.

```
import sikulilogger
def runTestSuite1(numberOfIterations):
  counter = 0

  while counter < numberOfIterations:
    for i in range(0, len(matrix)): # For each row in matrix
      success = assignIdentityToAllObjects(imgs[i]) # Assign all objects
                                                      # to imgs[i]
      if not success: # Exit if assignIdentityToAllObjects was unsuccessful
        log("Could not assign all. Exiting.", "error"))
        exit(1)

      for j in range(0, len(matrix[0])): # For each column in matrix row
        if matrix[i][j] == 0:
          continue # No action. Cannot assign object to the same identity
                   # as it already has.

        # Assign imgs[i] to imgs[j]
        success = assignIdentityToOneObject(imgs[i], imgs[j])
        # If assignment was unsuccessful, log an error and skip matrix row
        if not success:
            log("Could not assign. Row failed.", "error", imgs[j])
            break;

        # Check expected result on secondary screen
        if matrix[i][j] == 1:
          conflict(j) # A conflict exists, should be resolved by operator
        elif matrix[i][j] == 2:
          accept(j) # Automatically accept remote data and store as local
        else:
          reject(j) # Reject the remote data

    log("Iteration " + `counter` + " done!", "normal")
    counter += 1
```

Listing 6.1: Pseudo-code for Test suite 1 implemented in Sikuli.

Test suite 2: The second test suite had a similar setup to the first one, with a local and a remote system. In this test suite, the types of the objects were changed. The test aimed to verify that after the type of an object has been changed, this information was transmitted and acknowledged by both of the systems. Changing types is always an accepted action which should never result in a rejection or conflict. There are six object types (see Table 6.1), and since the action always is accepted, there will be six different test cases.

Listings 6.2 shows the pseudo-code for Test suite 2 implemented in Sikuli. In the code, `imgs` is an array containing images of each object platform, and `assignPlatformToAllObjects` and `assignPlatformToOneObject` are functions implemented to assign objects. From `sikulilogger`, the logging function `log` is imported.

```
import sikulilogger
def runTestSuite2(numberOfIterations):
  counter = 0

  while counter < numberOfIterations:
    for i in range(0, len(imgs)):
      # j is the index of the image to change from.
      # Change from the last image if i == 0
      j = i - 1 if i > 0 else len(imgs)-1

      # exp is the number of objects of this type we expect.
      # For the last iteration, we expect 10
      exp = 1 if i != len(platforms)-1 else 10

      noerrors = assignPlatformToOneObject(imgs[j], imgs[i], exp)
      # If assignPlatformOneObject failed, try to reset the
      # system to a default state (no statement)
      if noerrors == False:
        log("Failed changing to this platform, try reset", "error")
        success = assignPlatformToAllObjects(imgs[5], imgs, 10)

        #If reset action failed, exit
        if not success:
          log("Could not assign all. Exiting.", "error")
          exit(1)
        break

    counter+=1
```

Listing 6.2: Pseudo-code for Test suite 2 implemented in Sikuli.

6.2 Logging and reports

Unlike JAutomate, Sikuli does not have a built-in logging module to generate result reports. However, using Python, it is easy and straightforward to build a logging module. The one implemented in this project has a log method which will save the message it receives, the type of message (‘normal’, ‘error’, ‘debug’) and whether the operation should be considered successful or erroneous. Optionally, the user of the log method can also choose to take a screenshot of the screen, as well as highlight an area on the screen where the search for an image was performed. The screenshot functionality is a built-in function – `capture(region)` – in Sikuli, which takes a screenshot on the specified region, for instance a screen, saves the image in a temporary directory and returns the path. The screenshot functionality can only take a screenshot on one screen at a time. Thus several screenshots are taken if multiple screens are available. All screenshots are copied to same directory as the output log file. The screenshot functionality can also be obtained by using a separate program or Python library, which is called from the script. A script developer can choose to log at any time, and can log any type of message including, but not limited to, successful or failing image searches.

When consulting Saab AB employees who have carried out the tests, the conclusion has been that the behaviour of the system upon an error, as well as the steps leading up to the error are important; to lesser extent the behaviour of the system shortly after the error as well. The screenshot functionality of the logger is perceived as helpful in debugging the system upon failure. Thus, there is a need to log and save screenshots both of failing steps, and successful ones that precede and follows the failing steps. However, while the screenshots are needed for efficient debugging of the system, after a few days, there may well be tens of thousands of screenshots generated.

As described in Section 3.4, the tests carried out are loops of executing the same test suite repeatedly. Most of the present tests in Saab AB are fairly small, and/or consists of parts being independent from each other. This design means that every single execution step preceding an erroneous step is not important, usually only a few before or – as a worst-case – the preceding steps of that iteration. This allowed for a "garbage collector" of sorts that will later remove screenshots of successful steps that are not closely followed by failing steps. The script developer can tell the logger how many screenshots of successful steps before and after that will be kept.

The logger is efficient and will not clutter the user's harddrive with unneeded screenshots, and can often show exactly what went wrong in a test run. The main drawback is that it is not automatic, and the script developer will have to call the logging function in their code whenever something is to be logged. However, this is also the advantage of having a non-automatic logging function since the user is allowed to freely specify the extensiveness of the logging.

The logging module produces reports of the result of the run test suite, which is written to HTML files. One file for each type of message ('normal', 'error', 'debug') is written, which results in three different log files of different extent and content. The debug log is the most extensive log and contains all messages that are logged of all types while the normal log contains only messages of type 'normal' and 'error'. The error log only contains error messages. Thus, error messages are always considered more important than other types of messages and are included in all logs while debug messages are considered least important and are only present in the debug log. The advantage of having several output files is that if everything works as expected during the run and no errors are obtained, there is no need of clutter the log file with unnecessary information and the error logging file will be empty. However, if something goes wrong more extensive logging may be needed to find out what went wrong, then the debug logging file – which could be quite extensive – may be a good complement since it includes all log messages. Messages of different status are presented with different colours, for instance successful messages are presented in green while erroneous messages are presented in red. Since Python can output content to any text-based file format, it would be easy to change the output report to a different format if the need arises.

6.3 Modified OCR functionality

The OCR functionality is considered important as it expands what can be tested in the system. An example of where reliable OCR is needed, is if some result that is to be used in the test scripts is displayed in a non-machine-copyable form, e.g. in a text label. Another example could be if a specific cell or row is to be selected in a table widget based on the text in it, before further action is to be taken. A known problem in Sikuli is the poor performance of the OCR functionality, as it is not yet fully developed. The OCR functionality gives inconsistent results for the same input and sometimes fails all together. This is especially the case when reading text consisting only of digits, which is common in e.g. tables in the tested system. Due to the low accuracy, it was deemed too unreliable to be useful in our scripts in its current condition. As some test suites in the regression test verification specification need text recognition, a way to train the OCR engine was added to Sikuli. The improvements allows users to more reliably use the OCR functionality for their specific system. Whereas previously nothing could be done if the text could not be recognised, it is now possible to get more reliable results by using our extension to train the OCR engine from within Sikuli.

It was determined that the low accuracy of the OCR in Sikuli was because the letters in the tested system could not be recognised. Due to this issue, the use of a training function became a possible solution to improve the accuracy. The OCR functionality in Sikuli runs Tesseract as its OCR engine, which uses a machine learning approach to recognise different letters, as mentioned in Section 3.2.1. Thus, the low accuracy could be improved by training the engine on the letters that were not recognised correctly. Recent versions of Tesseract have a built-in training functionality which allows the user to create new training data files for the letters and the fonts that were not recognised. The new training data files can then be loaded into Tesseract to use when reading the text in the tested system, thus improving the results from previous attempts.

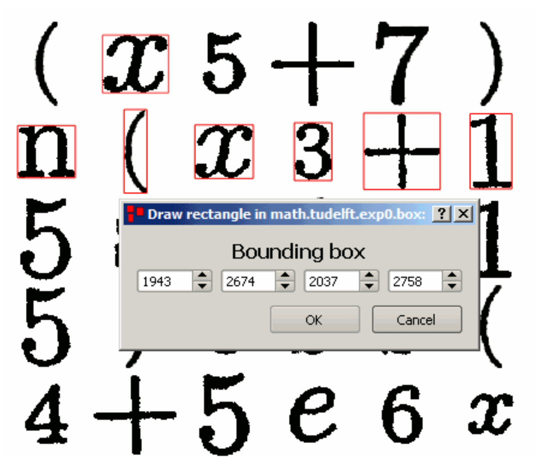
When training Tesseract on an image, it will attempt to interpret the symbols in the images as characters. Around the symbols in the image being looked at, Tesseract will place so called bounding boxes, which is a rectangle denoting what part of the image each character is in. A .box-file will be generated, which contains bounding boxes of all interpreted characters, and these bounding boxes are defined by their coordinates in the image. An example of a .box-file is given in Figure 6.1a, and a graphical representation of a different .box-file, taken from a screenshot of a .box-file editor, is given in Figure 6.1b. In Figure 6.1b, the bounding boxes are drawn as rectangles around characters, and a dialog is shown that lets the user change the bounding box.

By giving Tesseract an image for which one knows the correct text in, it is possible to utilise the training functionality to make Tesseract learn to recognise the font and characters. The generated .box-file will contain the characters found, and by matching against the correct text, it is possible to automatically correct any mistakes that were made in the training.

It was found that the OCR engine Tesseract performs well as a standalone application outside Sikuli, provided that the training data is good. However, the Windows

```
s 734 494 751 519 0
p 753 486 776 518 0
r 779 494 796 518 0
i 799 494 810 527 0
n 814 494 837 518 0
g 839 485 862 518 0
t 865 492 878 521 0
u 101 453 122 484 0
b 126 453 146 486 0
e 149 452 168 477 0
r 172 453 187 476 0
d 211 451 232 484 0
e 236 451 255 475 0
n 259 452 281 475 0
```

(a) The content of a small .box-file. The first column is the character detected, and the following four columns represent the coordinates of the bounding box of that letter.


Draw rectangle in math.tudelft.exp0.box:
Bounding box
1943 2674 2037 2758

(b) A graphical representation of a .box-file. Screenshot from the .box-file editor Qt-box-editor. [43]

Figure 6.1: A raw and a graphical representation of two .box-files

Sikuli installation is bundled with an older version of Tesseract which does not contain the training functionality. To be able to use the training opportunities, a standalone installation of Tesseract was installed as well. By calling this standalone Tesseract version with the commands shown in Table A.1 in Appendix A, a new training data file is created, which can better recognise the letters in the system.

The implemented training function in Sikuli uses the built-in training procedure in the OCR engine Tesseract together with an automatisation that evaluate if the provided data is usable for training and correct possible mistakes done by the OCR engine. After reading up on the quite undocumented source code of Sikuli, we found that both images and text internally are treated as regions in Sikuli, and the module that contains this logic was chosen to be extended with the training function. The design of the training function is shown in Algorithm 2 and described in the following paragraphs.

After consulting one of the Sikuli developers, a settings parameter was added to one of the core files in Sikuli which allowed the user to specify which training data file to use. We then extended Sikuli with a method `trainOCR(String imageOrText, String expectedText, String filename)` returning a Boolean when finished, telling whether the training process was successful or not. The method takes three arguments: an image of the characters to learn, a string of the identical text as in the image, and a name of the new training data file that is to be created. It is important that the text in the image and the text in the string is exactly the same, since the string is used in the training algorithm to determine whether it got the letters right or not.

The implemented training function allows the user to specify an image that was not correctly identified together with the corrected output, which this method uses to learn from. If Tesseract does not find all characters in the image, little can be done except providing a new and better image. However, if all characters are found, a correct classification of each character can be achieved by editing the characters that were not correctly classified in the .box-file and then generate the training data file.

The implemented training functionality works in the following way: If the input was not as expected or needed files could not be accessed, the algorithm quits (lines 1-5 in Algorithm 2). After that, the algorithm runs the Tesseract command to create a .box-file (line 9). If the number of lines in the .box file is not equal to the number of characters specified in the expected text parameter, the algorithm cannot continue for the given input, and will therefore end (lines 12-14). Otherwise, the .box-file will be checked and each character that differs from the corresponding expected character is replaced (lines 16-22). Then the other Tesseract commands are run, which together creates the training data file (line 24-33). This file is then moved to the Tesseract data directory, and a clean-up is performed, deleting all auxiliary files (lines 34-45).

Algorithm 2: Our implemented training function, trainOCR, for the OCR functionality in Sikuli which uses the built-in training function for Tesseract (according Table A.1 in Appendix A).

```
Data: $imageOrText$: image to recognise, $expectedText$: expected text contained
      in image, $filename$: name of created training data file
Result: Boolean (successful or not)

$tessdataDir \leftarrow$ tessdata directory (/libs/tessdata)
if $imageOrText == null$ or $expectedText == null$ or $!tessdataDir.exists()$ or
$filename.exists()$ then
    return False
end

$n_{Text} \leftarrow$ number of characters in expectedText
$imgNoExt \leftarrow$ imageOrText without file extension
$boxfile \leftarrow$ tesseract $imageOrText$ $imgNoExt$ batch.nochop makebox
$n_{Img} \leftarrow$ number of characters in boxfile

if $n_{Img} \neq n_{Text}$ then
    return False
end

forall the $line$ in $boxfile$ do
    $c1 \leftarrow$ first character in $line$
    $c2 \leftarrow$ corresponding character in $expectedText$
    if $c1 \neq c2$ then
        Replace $c1$ in $line$ with $c2$
    end
end

$trfile \leftarrow$ tesseract $imageOrText$ $boxfile$ nobatch box.train
$unicharset \leftarrow$ unicharset_extractor $boxfile$
write "$imgNoExt$ 1 0 0 0 0" to font_properties

shapeclustering -F font_properties -U unicharset -O $unicharset$ $trfile$
mftraining -F font_properties -U unicharset -O $unicharset$ $trfile$
cntraining $trfile$
Rename unicharset, normproto, inttemp, pffmtable, shapetable with $filename.$ as
prefix

combine_tessdata $filename.$
Move $filename$.traineddata file to the tessdata directory
Delete all other generated files
return True
```

7

Results

The evaluation and comparison in Chapter 5 resulted in that Sikuli was considered to be the tool most suited to Saab AB's needs. A summary of the comparison is given in Section 7.2. Due to the choice of tool and the poor results of the OCR functionality, Sikuli had its OCR functionality improved, as was explained in Section 6.3. This enabled training on images specified by the user. The results from running the OCR functionality in Sikuli before and after this modification are presented in Section 7.1.

7.1 Training functionality for OCR

The performance of the modified OCR functionality was tested by running a few tests and measuring the accuracy for those. The tests were performed both when using the original OCR functionality in Sikuli, without additional training, and when using the OCR functionality together with training on the used interface. The testing of the OCR functionality consists of reading some of the non-animated areas containing text in the tested system. This includes menus and tables available in the interface. The accuracy for the test is determined by how many of the text strings in each test is correctly recognised and how many times, i.e. if one string out of five was recognised correctly in all iterations, this gives an accuracy of 20%. A string consists of between 2 and 21 characters. An accuracy of 0% means that none of the strings where correctly recognised, but this does not mean that all characters in that string was incorrectly recognised. If a string consists of 21 characters and one of them is incorrectly identified, this string failed and will be given the accuracy 0%. This determination for measuring the accuracy was decided since the usability of the OCR functionality in this case depends on its ability to read words rather than just the single characters. However, when only a few characters were incorrect this is included as a comment. We also include two accuracy measures, one that counts missing spaces between words as a correct interpretation and one that does not. Neither of the accuracy measures make any difference between upper case and lower case letters. Three different tests are chosen – A, B and C – which are described below:

Test A Read the menus. There are five of these and some of them have an underscore under a letter (indicating a keyboard shortcut).

Test B Click on one menu and read the six different sub-menu items.

Test C Read three rows in a table, all having three columns with data (both letters and digits).

The tests above were chosen by us, since there were no such tests in the regression test verification specification. However, it may be included in the future and to be able to include text in the tests the OCR functionality must be tested as well. The tests were carefully chosen to represent different parts of the tested interface with different size, style, background colour and placement. Thus, we are able to test the OCR functionality on several different parts of the interface.

For comparison, the accuracy for the OCR engine – Tesseract – was measured as well as a stand-alone application. The results for this are shown in Table 7.1.

7.1.1 Accuracy for Tesseract

	Test A	**Test B**	**Test C**
Accuracy, without additional training:	20%	0%	11%
Accuracy, with additional training on Test A:	60%	0%	11%
Accuracy, with additional training on Test B:	20%	50%	11%
Accuracy, with additional training on Test C:	20%	0%	89%

Table 7.1: Results for Tesseract as a stand-alone application (10 iterations).

The accuracy for Tesseract without additional training is pretty bad for the three tests, which is seen on the first row in Table 7.1. Only one menu of five was interpreted correctly (Test A) and none of the menu items were (Test B). In the table test (Test C), only one column on one row was correctly recognised. With additional training on the menus in Test A, three of the menus were correctly interpreted. However, the other tests were unaffected. The reason why the other two menus in Test A were not recognised correctly is because Tesseract did not find all the characters, which makes the training process on these unsuccessful, as previously stated. With additional training on the menu items in Test B, only one of the menus were recognised correctly, and the results for Test C remains the same. However, Test B was interpreted correctly for three of six menu items. One item was almost correct, but Tesseract missed a space and the other two were not possible to train on. With additional training on the tables in Test C, the results for Test A and Test B remains the same as without training. For Test C, all the elements in all the three rows in column 1 and column 2 were interpreted correctly. However, none of the rows were correctly interpreted for column 3 which is due to an unsuccessful training process for these.

7.1.2 Performance for OCR – without additional training

The accuracy for running the built-in OCR functionality in Sikuli with only the provided training, actually gives a better result than Tesseract got as a stand-alone application. This is shown in Table 7.2.

	Test A	**Test B**	**Test C**
Time:	10min 46s	1h 16min 18s	33min 1s
Accuracy, excl. missed spaces:	40%	33%	67%
Accuracy, incl. missed spaces:	40%	67%	67%

Table 7.2: Results for OCR, without additional training (1000 iterations).

The accuracy is about 40% for Test A, 33% (or 67% if missed spaces are not counted) for Test B and 67% for Test C. For Test A, one menu is totally misinterpreted while two others only had one letter wrong. In Test B, one menu item is totally misinterpreted as well and one got a letter wrong and missed a space. Two other items also had missed spaces. In Test C, the first column is only interpreted correctly once and the second column is misinterpreted once. The third column is correctly interpreted for all rows. In the misinterpreted rows, only one character is wrong in the first and second column.

7.1.3 Performance for OCR – with additional training

The training for the OCR functionality was performed in three steps, one for each test. Table 7.3 to Table 7.5 shows the result running the OCR functionality on each of the tests with training on each of them. In Table 7.3 the OCR functionality is trained on Test A, in Table 7.4 it is trained on Test B and in Table 7.5 it is trained on Test C.

	Test A	**Test B**	**Test C**
Time:	12min 52s	1h 19min 13s	35min 24s
Accuracy, excl. missed spaces:	60%	33%	89%
Accuracy, incl. missed spaces:	80%	67%	89%

Table 7.3: Results for OCR, with additional training on Test A (1000 iterations).

The accuracy for running the OCR functionality with additional training on Test A is 60% for Test A (or 80% if missed spaces are not counted), 33% (or 67% if missed spaces are not counted) for Test B and 89% for Test C. The reason for the accuracy for Test A not being 100% is since two of these is not possible to train on, as explained for Tesseract in Section 7.1.1. The two menus being misinterpreted are due to one letter being lower case when it should be upper case and one letter is missing. The result for Test B is unaffected by the training while the result for Test C is better since both the first and the third columns are interpreted correctly for all rows.

	Test A	**Test B**	**Test C**
Time:	15min 3s	1h 22min 47s	36min 18s
Accuracy, excl. missed spaces:	60%	33%	89%
Accuracy, incl. missed spaces:	80%	67%	89%

Table 7.4: Results for OCR, with additional training on Test B (1000 iterations).

The accuracy for running the OCR functionality with additional training on Test B is 60% for Test A (or 80% if missed spaces are not counted), 33% (or 67% if missed spaces are not counted) for Test B and 89% for Test C. This is the same result as with training on Test A. The reason for the accuracy for Test B not being higher is because two menu items cannot be recognised by the trainer and is thus not possible to train on, as explained for Tesseract in Section 7.1.1. For the recognised menu items in Test B, two spaces are missing, and in Test A, one upper case letter is misinterpreted for a lower case letter and one letter is missing. In Test C, only one row is misinterpreted in the second column while the other ones are correct.

	Test A	**Test B**	**Test C**
Time:	14min 38s	1h 22min 4s	36min 13s
Accuracy, excl. missed spaces:	60%	33%	100%
Accuracy, incl. missed spaces:	80%	67%	100%

Table 7.5: Results for OCR, with additional training on Test C (1000 iterations).

The accuracy for running the OCR functionality with additional training on Test C is 60% for Test A (or 80% if missed spaces are not counted), 33% (or 67% if missed spaces are not counted) for Test B and 100% for Test C. The results are quite similar to the other two results with training. However, the difference is that the result for Test C is correct for all three columns and all three rows.

7.2 Tool recommendation

The tool comparison, described and discussed in Section 5.4, resulted in that the VGT technique was considered very valuable, but it is new and thus neither of the evaluated tools are as mature as they could be. The tools were similar in many aspects, but Sikuli was considered more suitable for the tested system and is thus recommended to Saab AB. The complexity of the tested system, with moving objects, meant that the scripts needed to have high fault tolerance. This was easier and faster to acquire in Sikuli due to the powerfulness in the scripting language. Despite some bugs, the clearness in the editor made it easy and fast to write new scripts. The value of using the VGT technique for regression tests of the tested system was very clear and much more beneficial than manual testing. Due to the benefit of the VGT technique and the result of the tool comparison, Sikuli is recommended to Saab AB as a suitable testing tool for their regression testing of the system.

Several bugs of various impact were found in the tested system, both known and unknown. For instance, some buttons did not have the expected behaviour. One such example is the button used to centre the map on the object currently selected. Sometimes repeated presses of the button would toggle between correctly centring on the selected object, and centring on the default position on the map. Another example of a bug that was found during the testing was that sometimes the right-click menu, which is used for assignment, became inactive and thus unusable. This renders the operator effectively unable to interact with the object; in other words a serious bug indeed. Finally it was discovered that the system sometimes was unstable when running the test for a long time, which made the system lose connection. This is of course also severe, and a known issue. However, it should be noted that the tested interface, and system, is constantly under development, and was updated several times during the work process. Since the system is not a stable release, bugs are to be expected. The system also suffered from some memory leaks during the process, some of which were discovered during our testing.

8

Discussion

As stated in the introduction, this thesis aims to evaluate some VGT tools and give a recommendation on how and whether to use VGT for regression testing. This cannot be properly done without a cost/benefit analysis of the technique, which is presented on the following pages. Also, an analysis of the evaluation and the modifications done to the OCR functionality is included on the following pages.

8.1 Implementation differences and moving objects

The implemented tests may differ somewhat between the tools and the test verification specifications, as stated in Section 5.4.2. Since the test verification specifications are written in an informal language and need to be translated into a more formal language, i.e. implemented, to be runnable in a tool. Due to the features of each tool and the complexity of the tested system, the implementations may differ, although they still test the same functionality.

In Section 5.4.4, moving objects and the problems they pose are described – as well as ways of mitigating them. There are plenty of scenarios where the ways of mitigating the problems described in Section 5.4.4 might not be possible, and if the mitigation techniques mentioned there do not work, the test cannot be performed. However, we found that there are often ways to work around these limitations by adapting the tests to be conducted to the VGT technique.

By focusing on what functionality of the tested system was supposed to be evaluated by the regression test verification specification, we found that it was often possible to write scripts that were similar rather than identical to what the test verification specification directed, but still evaluated the same functionality of the tested system. In our tested system, a recurring operation would be to perform some action on one out of several identically looking objects, and making sure that a different computer connected on the network saw the action, as described in Section 6.1. If the tested system has multiple displays with the same zooming and map region visible, such that the screens are identical, it could be possible to take the region on the screen where a change is made, and then check the same region on the other screen for the change. However, this requires the screens to have identical zoom level, and that we can anticipate the movement of the objects and compensate for this if needed. A much easier problem to solve is to verify that one object on the other computer reacted as expected rather than verifying that the correct object reacted. This is much less complicated and is expected to give a result that is good enough. The complication arises when the script wants to verify that the action took place as expected, the movement of the objects results in that the script no longer knows where the object is, and it might have several identically looking objects to choose from. Thus, taking a region and checking the corresponding region on the other screen might not always work. This is because if the object to be found and confirmed on the other screen moves away before it has been verified and the next object looks alike and moves to the same position, the VGT tool might verify the wrong object. In such case, solving the less complicated problem of verifying the number of objects expected rather than the right object is a better approach.

Based on the results from running our implemented test suites, the VGT technique is suitable for both system testing and acceptance testing. But, failure mitigation must be added to handle the moving objects and compensate for slowness in the system and the tool. This affect the ability of the technique to simulate an end user but this is in fact a problem for a human user as well since even a human will miss to click on a moving object sometime. Thus, we do not consider this a big problem and we consider the VGT technique applicable to simulate an end user in an animated environment as well as a non-animated.

8.2 Informal analysis of perceived cost/benefit

Visual GUI testing is a rather new technique which has not yet been used extensively in an industrial context, but we are convinced that it will be in the near future. Neither manual nor automatic testing can guarantee that all errors will be found, but the probability of finding an error increases with the testing. The greatest benefit of automatic testing and visual GUI testing is raised test frequency and that the tests may run without any human interaction, for instance during nights and weekends. This may result in better and faster feedback to the developers. However, as stated in the beginning of Chapter 3, automatic testing should be seen as a complement to manual testing. Above all, exploratory testing cannot be automated efficiently yet. Visual GUI testing only mechanically runs its test scripts, so any errors that do not show up there won't be found. Such errors can be revealed by manual exploratory testing though.

All of our observations and conclusions come from testing only parts of the system, as only a subset of the regression test verification specifications were implemented. We prioritised getting extensive data from a few implemented tests, rather than getting less data from more tests. It may not be possible to automate the whole test verification specification, due to differences between the test and what is applicable in the VGT tool. This was briefly discussed in Section 8.1, where we discuss that the tests needed to be adapted to the VGT tools rather than strictly follow the test verification specification. Another example is that most tools cannot interact with all parts of the system; for instance, Sikuli has no built-in way to test sound. The number of test suites that can be automated is in proportion to the profit – the more automated cases, the better. Even if 100% coverage is not possible, the technology is still beneficial. Any test suites that cannot be automated should be part of the manual testing that still needs to be conducted.

Since we chose the open source tool Sikuli after the evaluation presented in Chapter 5, there is no cost to get the tool. However, there is always a development cost for the scripts as mentioned in Section 5.4.5, as well as a maintenance cost. These are very hard to measure, and should be evaluated during a substantial amount of time. Thus, we can only speculate in how high these costs are. To some extent, this is mentioned in Alégroth's work [2], where the estimated development time and maintenance time is found to be less than the time spent manually performing the test, just after a few runs. As the advantage of automated testing is to be able to run tests continuously (for instance during nights and weekends), the cost for developing and maintaining the scripts are marginal with respect to the benefit. There will be an initial cost of developing the test suites, as the tests have to be implemented. However, the cost for performing the tests – which is one of the biggest costs today – will decrease to almost zero. The costs of development, implementation and maintenance need to be investigated during a substantial period of time. If those costs turn out to be too high in respect to the gain of more extensive regression testing, the visual GUI testing technique will become obsolete in an industrial context. Our assessment is that visual GUI testing has a lot of potential to last, and that the advantages outweigh the disadvantages but this needs to be proven by further research.

As stated in Section 8.1, we encountered some problems with moving objects, which was solved by writing scripts similar rather than identical to what the test verification specification specified. While this might seem like it has serious implications for what is testable, we argue that it is not entirely different from what a human testing this manually would experience. If many tests are to be carried out manually, the human error often leads to negligence of the details; a stressed and/or tired human testing might perform an action on an object, and simply acknowledge that the expected result was visible on the networked computer, rather than strictly verifying that the expected result happened on the correct object. If the tasks are of repetitive nature, we believe that humans will tend to perform poorly on these tests, something that Saab AB employees who have carried out manual testing also independently confirm. Humans generally tend to let their mind wander during repetitive tasks, leading to attentional disengagement, something that will occur despite optimal training, competence and motivation of the performer [44]. As Smallwood and Schooler notes in their article The Restless Mind:

> *"In mind wandering, it seems that the automatic activation of a pertinent personal goal temporarily overshadows the more immediate goal of task com-*

pletion. Such goal switching may be enabled by the tendency for experience to become decoupled from meta-awareness, such that individuals temporarily fail to notice that their task-related processing has been hijacked by a more personally relevant goal." [45]

Smallwood and Schooler also note that verbal reports have indicated that between 15% and 50% of a participant's time is spent mind wandering across a diverse variety of tasks [45].

8.3 Tool evaluation

The evaluation and comparison between the tested tools was more thoroughly discussed in Section 5.4. The comparison was done according to the tools properties, performance in the implemented tests, fault tolerance and the ability to handle moving objects. To some extent, the cost and time aspects were discussed as well since these are important aspects of a sustainable tool. The tool properties are very similar to the static tool properties presented in Alégroth's work [36], and the results presented here are similar as well. It would have been desirable to use a statistical method to analyse the collected quantitative data to see if there is any statistical significant difference between the tools. This could be done by using for instance the Wilcoxon test [46], but was not conducted due to lack of time.

The ability to handle moving objects and fault tolerance was considered important in the evaluation, due to the complexity of the tested system. This was much easier to handle in Sikuli, especially when having prior knowledge about programming. This prior knowledge likely affected the tool recommendation; the scripting language in JAutomate was very intuitive but the one in Sikuli has much higher expressivity – something very valuable for someone with programming experience. Those with less experience may have reasoned differently. It should also be noted that the experience of a tool is individual, and thus different people may prefer different tools.

More focus on the time and cost aspect would have been desirable, but as stated before, it takes a long time to achieve the needed data in this area. It is also hard to collect general data for areas like development costs, since it is affected by many personal aspects such as prior knowledge and experience. Thus, we have not focused on that area, and it is suited for future studies.

As stated in the introduction, there are several different VGT tools on the market; both proprietary and open source. To extend the report further, it would have been desirable to extend the evaluation and comparison with more tools that could be suitable for this kind of tested system. An example of such a tool could be eggPlant. This would give more general results by evaluating a larger part of the VGT market and thus a more accurate evaluation of the VGT technique. It was not conducted due to lack of time.

As per the discussion in Section 8.1, the implemented tests differ somewhat between the tools, so the running times in Table 5.5, 5.6 and 5.7 must be compared with consideration to this aspect. Also, the number of failing tests, image recognition failures and misses to click on a moving object in Tables 5.6 and 5.7 must be carefully compared, as the number of executed test iterations and number of interactions with an object differs substantially between the test suites. For instance, Test 1 is much larger than Test 2 – which is evident from the big difference in duration in relation to number of iterations. A high number of image recognition failures and misses to click on a moving object is not necessarily a bad result, if the code can handle the failures. The measures described in Section 5.4.3 and 5.4.4 should be implemented in the code, making it is more robust and better at handling errors.

It should be noted that both the tools and the tested systems have had parameters optimised, most likely affecting the results, as mentioned in Section 5.4.2. We noted for instance that a slow mouse cursor speed could pose some problems when interacting with the moving objects. The options to display additional visual information next to objects were turned off, increasing image recognition results. The similarity parameters for the image recognition needed to be adjusted to suit the tested system and different values could be needed in different situations. However, the values of the parameters may not have been optimal. While some simple trial-and-error often works for quickly finding parameters that work, it may be very time-consuming to determine optimal values.

To get more general results in terms of performance in the evaluation and to raise the validity of the results, it would be desirable to implement more test suites to test more functionality in the system. Two test suites is few in relation to the number of available test suites in the regression test verification specification. However, the implemented test suites were carefully chosen by Saab AB to be representative for a large part of the regression test verification specification. The test suites are also big and advanced in the aspect that most of the actions performed in the tests are performed on moving objects. It should also be noted that the two implemented test suites in fact consists of 49 respective six test cases, which raise the validity and generalisability.

For Test 1, only 100 iterations were performed since each iteration took about half an hour to perform, due to the extensiveness of the test suite. It would also be desirable to perform more iterations of the tests to get more accurate data and validity, especially for Test 1. Had there been more time, more iterations of each test suite for each tool would have been performed. We would like to conduct about 10 000 iterations, since by then, all the random events that may occur has most probably occurred at least once and any divergent results do not affect the overall result very much since it most likely heads towards an average. This is said to happen after 10 000 iterations, which then increases the validity of the results.

We categorised the different sources of faults in Section 5.4.3. Many of those could efficiently be mitigated by handling unexpected events in the script. However, there may be other faults that we have not found yet and our implemented scripts do not handle all possible faults. Especially external interference is unfeasible to be robust against, but due to the low likelihood of it happening, this is not considered a problem. For a large system, the possible events causing unexpected system behaviour may also be multiple, therefore demanding robustness of the scripts.

8.4 Modified OCR functionality

As stated in Section 6.3, a training function was added to Sikuli, in order to improve the results of the OCR functionality. This allows the user to train the OCR engine on text in the tested system that was not correctly recognised. This modification improved the accuracy of the OCR functionality as shown in Section 7.1, but not as much as we had hoped.

The reason for not getting the expected accuracy is partly due to the fonts we wanted to train on. These are quite small with little space between the characters, which makes it hard to distinguish the characters from each other – even for a human. This resulted in that we were unable to train on several strings since their characters were not recognised by Tesseract. This is the main limitation for Tesseract and this technique. If the characters to train on are not found in the first place, it is impossible to train on these. Due to this limitation, it is not possible to archive an accuracy of 100% for all test cases since our training data is bad in the tested system.

The other reason for the unexpected accuracy is that the OCR functionality in Tesseract and the OCR functionality in Sikuli do not fully seem to follow each other. This is a bit odd since Sikuli uses the Tesseract engine for the OCR functionality, and thus it would be reasonable that those two gave the same results. The results may differ due to that the integrated Tesseract engine is still experimental in Sikuli and it is thus not finished functionality. However it is still a bit odd that Sikuli got better results with training than Tesseract did. The effect of the training is much bigger with Tesseract than with Sikuli. We have not had time to investigate this further, and as stated before the OCR functionality in Sikuli is still experimental. Thus, this problem may be solved when this functionality is fully developed. Before the OCR function is fully developed, we cannot confirm the real improvement of our algorithm and the results in this state is not good enough for the tested system. The accuracy may increase with the OCR function being completed in Sikuli, and then it may be applicable on the tested system. The OCR function together with our training function is also applicable for systems with clearer fonts and bigger characters, since this increases the accuracy.

Even though the accuracy did not improve as much as we hoped, our implemented training functionality is still an improvement that works as expected. This could especially be seen on the results for just running Tesseract in Table 7.1, where the extra training improved the results significantly (with respect to the limitation of the difficult font). The results for running Tesseract as a stand-alone application were only used for proving the concept of training an algorithm. Thus, only 10 iterations were performed but the result was the same for all of them. This was also the case with Sikuli for 1000 iterations, which may be due to that the OCR algorithm in Tesseract – which is used in Sikuli as well – may be deterministic.

As mentioned in Section 7.1, a stand-alone installation of Tesseract is needed as the one included in Sikuli lacks the training functionality. Most likely, Sikuli will be updated to include the newer version of Tesseract in future releases, so while the inconvenience of having to install Tesseract separately to use our modified OCR functionality may be an annoyance, we believe it is a temporary problem. For other operating systems than Windows, Tesseract is currently not included in the Sikuli installation, further supporting our approach.

All OCR algorithms which are based on machine learning can use this training approach. However, the training function must be modified to fit the used OCR engine since the implementation and commands used are Tesseract specific.

To improve the accuracy of the OCR function even further, a possibility is to try to improve the pre-processing of the images done in Sikuli. By pre-processing the images to get the text to read clearer and bigger, it may be possible to help Tesseract to better recognise difficult fonts and characters to achieve a higher accuracy.

9

Conclusion

In this thesis, two visual GUI testing tools – Sikuli and JAutomate – are evaluated for GUI testing purposes on a system at Saab AB. As concluded in the tool comparison in Chapter 5, the accuracy is good – and quite equal – for both of the tools and the overall performance is satisfying. Due to the complexity of the tested system, fault tolerance and the ability to handle animated objects are important properties for a suitable tool. Both of the tools are able to handle this but the powerfulness and expressivity in Sikuli's scripting language made Sikuli able to handle this in a better way. JAutomate lacks functionality needed to create more advanced scripts, such as data structures like arrays, methods able to return values and efficient expression evaluation. This affects the expressivity of the scripting language, and we feel that issues with fault tolerance and object movement have to be kludged around. Thus, Sikuli is determined better suited for the tested system due to the powerfulness of the scripting language and is recommended as the most suitable VGT tool – based on our tool comparison. It should though be noted that Sikuli was considered most suitable for this specific system and Saab AB's need, and it is thus not a general conclusion that Sikuli always would be a better choice.

The thesis work also included investigating if the accuracy of the OCR functionality in Sikuli, which was poor, could be improved. We can conclude that this can be achieved by implementing a training function for the OCR functionality in Sikuli, using the OCR engine Tesseract. This modification gives the user the opportunity of helping the OCR functionality to recognise the wanted text. The user is allowed to give images and expected output to the OCR training functionality to improve the text recognition for their needs. The evaluation of the results from running the original OCR functionality in Sikuli and running our modified OCR functionality with additional training shows that the latter improved the accuracy. But, it did not give the expected accuracy improvement. Nevertheless the results were improved and the unexpected inaccuracy was due to limitations of Tesseract and the training process, and due to the unclear and small font we attempted to read. Due to these limitations, the accuracy cannot reach 100% for all tests which partly answers our research question about how much improvement is possible. Our implementation works as expected and serves as an important part of the OCR functionality in Sikuli. Since the accuracy of course is important for the function to be usable, this training functionality improves the OCR usability significantly. However, the OCR functionality itself in Sikuli is still experimental and not fully developed so the real improvement of this modification of the tool cannot be confirmed.

Automated testing is an important part of the development process of software products and is widely used for lower levels of system, such as unit tests. Nevertheless, also the highest level – i.e. the GUI – must be tested, which is often carried out manually. This is not only costly and time-consuming but also error-prone, which motivates the need for automated GUI testing as a complement. There are some other automated GUI testing techniques available, such as widget based GUI tools. However, these suffer from limitations and drawbacks that make them unsustainable in the long-term and affect their usability. Visual GUI testing is a quite recent technology. It builds on image recognition, which is an area that is continuously expanding and developing. The experimental results from the tool comparison where the performance of the tools is tested, provides a technical evaluation of visual GUI testing using image recognition concluding that it is a very promising technique. As informally discussed in Section 8.2, this is due to the powerful concept of image recognition making it unaffected by changes in the GUI layout, code or API, which were the primary drawbacks of the widget based GUI tools. However, visual GUI testing needs more evaluation; especially over time to evaluate the maintenance and development costs. Other studies indicates that these costs are low relative to the benefits, but since this aspect is still missing we cannot fully answer the question of the costs relative the benefits more than that our results and other studies indicate that the benefits are high relative the costs. It must also be noted that automated testing in general cannot fully replace the need of manual testing, since especially exploratory testing is still needed and is an important part of the testing process. The analysis also points on a limitation regarding what can and cannot be automated. As discussed in Section 8.2, the tools themselves have some limitations in this aspect, and it must also be noted that there is a difference in what is stated in a verification specification and what can be implemented; especially when moving objects are involved.

The experimental results from the tool comparison gives a technical evaluation of the robustness of using visual GUI testing for animated objects. The results show that the visual GUI testing tools can handle the moving objects, but they might pose problems that must be handled. This is due to the objects not being in the same location when an action is to be performed, as they were when the image recognition algorithm located them. Some ways of mitigating the problems were found, which may or may not be feasible in practise by a company, depending on their specific system to test. The main conclusion to draw for the tested system is that the tests to be performed often can be reformulated to work around the problem. Rather than strictly following a test verification specification, the script implementation could, in all the implemented test suites, be adapted to test the same functionality, without being stopped by the problems that the movement caused. It can thus be concluded that in an industrial context, the tests might have to be adapted for the VGT technique to be applicable. Almost every image recognition failure and every interaction failure in the tested system is solved by retrying the action, which also is the simplest and most general way of solving the issue. The gain of using this approach is shown in Figure 5.5 in Section 5.4.4.

Failure mitigation was discovered to be very important to get runnable test scripts. If not mitigated, it sometimes resulted in the whole script failing. During this work, five different types of faults were found to occur more or less often using visual GUI testing tools. This is due to the complexity of the system, which raised several different failures where many of them were due to the movement of the objects. The most frequent failure was that when the tool tried to interact with an object, it could have moved from the location on the screen where it was found. The other less frequent failure was that the image recognition failed. We found that these types of failures could simply be solved by retrying the action, which solved most of the failures that we got. The failures left are mostly very difficult to predict and they often have a serious impact on the system; thus we decided that the most suitable way to handle them is to terminate the script.

9.1 Answering the research questions

To summarise, two visual GUI testing tools are evaluated for GUI testing purposes on a system at Saab AB. Some robustness issues, mostly due to moving objects, are found which could limit the use of the tools. This answers our first research question about the applicability of visual GUI testing tools as well as their robustness and limitations. The robustness issues are related to the presence of moving objects in the system that pose problems. However, the found problems can be mitigated by fault tolerance which answers the question about possible problems with moving objects. Furthermore, five different types of faults that could occur were found, and also some ways of mitigating them, which answers the question about what kind of faults can occur. Another issue found during the evaluation was the poor accuracy of the OCR functionality in Sikuli. The accuracy is improved by adding a training function, however not as much as we hoped. It is determined that an accuracy of 100% is not possible due to the tested system and limitations in the underlying OCR engine. This answers our research question about the applicability of a training function and partially how much improvement is possible and limitations. There may be more limitations that we did not have time to investigate, and the maximal improvement possible is hard to determine since the OCR function in Sikuli is still experimental. The last question concerning perceived costs relative to the benefits has been only partially answered. This is due to the fact that though we have been measuring time in our test experiments, we have not performed an exhaustive and precise cost analysis on other costs but rather made an informal analysis based on our understanding on how users might experience interacting with the tools.

9.2 Future work

Since visual GUI testing is a rather new technique, there is a lot to explore in the area, especially in an industrial context. It has not been adopted by many companies yet, but due to the benefits of this technique we speculate that it is probably just a matter of time before it is. Moving objects, which was mentioned in this thesis, is a rather unexplored area and can be further investigated. The limitations of visual GUI testing applied to non-static interfaces would be an interesting topic.

The difference in accuracy between running Tesseract through the OCR function in Sikuli and as a stand-alone application should be further investigated. This may be a result of the OCR function still being experimental in Sikuli, which is why this function also needs to be completed. Improvements to the training function used in Tesseract may also contribute to a higher accuracy.

Finally, more research is needed in the areas of development cost, implementation and maintenance. Since this needs to be done during a substantial period of time, it is a major area which will take a lot of time investigating. VGT is probably a lasting technique, where the benefit is much higher than the costs, but this must be proven through extensive research during a long time.

Bibliography

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

[2] Alégroth, E. *On the Industrial Applicability of Visual GUI Testing*. Thesis for The Degree of Licentiate of Engineering , No 100L, Department of Computer Science & Engineering, Division of Software Engineering, Chalmers University of Technology and Göteborg University [2013].

[3] TestPlant. eggPlant for Defense & Aerospace: Defense Software Testing [2014]. Visited: 2014-03-10.
URL: *http://www.testplant.com/eggplant/testing-tools/eggplant-developer/*

[4] Memon, A.M. and Soffa, M.L. Regression testing of GUIs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 28(5). ISBN 1581137435 [2003] pp. 118 –127.

[5] Olofsson, A. *Real time Signal Processing for Airborne Low-Frequency Ultra Wide-band SAR (In Swedish)*. Master's thesis, Chalmers University of Technology [2003].

[6] Hollmann, M. Christian Huelsmeyer, the inventor [2007]. Visited: 2014-08-07.
URL: *http://www.radarworld.org/huelsmeyer.html*

[7] Skolnik, M.I. *Radar handbook*. McGraw-Hill's AccessEngineering. McGraw-Hill, 3rd ed. [2008].

[8] Wiora, G. Sonar Principle [2005]. Visited 2014-03-10. Licensed under the Creative Commons Attribution-Share Alike 2.5 Generic license.
URL: *https://commons.wikimedia.org/w/index.php?title=File:Sonar_Principle_EN.svg&oldid=105775065*

[9] Shaw, A. Command, Control and Communications. *Scientia Militaria: South African Journal of Military Studies* [2012] vol. 132. ISSN 2224-0020. doi:10.5787/10-3-700. Visited 2014-05-15.
URL: *http://scientiamilitaria.journals.ac.za/pub/article/view/700/704*

[10] Department of Defense. Dictionary of Military and Associated Terms [2010]. Visited: 2014-05-15.
URL: *http://www.fas.org/irp/doddir/dod/jp1_02-april2010.pdf*

[11] Onoma, A., Tsai, W.T., Poonawala, M. and Suganuma, H. Regression Testing in an Industrial Environment. In *Communications of the ACM*, vol. 41(5). ISSN 0001-0782 [1998] pp. 81 – 86.

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

[13] Alégroth, E. Random Visual GUI Testing: Proof of Concept. In *SEKE* [2013] Visited 2014-07-10.
URL: *http://www.cse.chalmers.se/˜algeroth/publications/Random_vgt_Alegroth_2013.pdf*

[14] Chang, T.H. Using Graphical Representation of User Interfaces as Visual References. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM. ISBN 1450310141 [2011] pp. 27 – 30.

[15] Sjösten-Andersson, E. and Pareto, L. Costs and Benefits of Structure-aware Capture/Replay tools. In *SERPS'06* [2006] pp. 3 – 12. doi:10.1.1.103.4768. Visited 2014-07-10.
URL: *https://www8.cs.umu.se/˜jubo/Papers/SERPS06.pdf*

[16] Gonzalez, R.C. and Woods, R.E. *Digital Image Processing*, chap. 12. New Jersey: Pearson Education, 3rd ed. [2008] pp. 861–906.

[17] Russell, S.J. and Norvig, P. *Artificial Intelligence A Modern Approach*. Prentice Hall series in artificial intelligence. Boston: Pearson Education, 3rd ed. [2010].

[18] Perceptron - Wikipedia, the free encyclopedia [2014]. Visited: 2014-07-01.
URL: *http://en.wikipedia.org/w/index.php?title=Perceptron&oldid=614511057*

[19] Ljunglöf, P. Artificial Intelligence, Project 1 [2013]. Visited: 2014-07-01.
URL: *http://www.cse.chalmers.se/edu/year/2013/course/TIN171/learning_project.html*

[20] Qwertyus. Scatterplot of a synthetic binary classification dataset, with the decision boundary of a linear support vector machine (svm). [2013]. Visited 2014-05-15. Licensed under the Creative Commons CC0 1.0 Universal Public Domain Dedication.
URL: *https://commons.wikimedia.org/w/index.php?title=File:Linear-svm-scatterplot.svg&oldid=124054916*

[21] Shapiro, L.G. and Stockman, G.C. Computer Vision. *Scitech Book News* [2001] vol. 25:pp. 279–325.

[22] Wolfram Research, Inc. Cross-Correlation – from Wolfram Mathworld [2014]. Visited 2014-04-24.
URL: *http://mathworld.wolfram.com/Cross-Correlation.html*

[23] The MathWorks, Inc. 2-D cross-correlation - MATLAB xcorr2 - MathWorks Nordic [2014]. Visited 2014-07-04.
URL: *http://www.mathworks.se/help/signal/ref/xcorr2.html*

[24] Sjöblom, J. Government Building of Tuzla Canton burning during the Bosnian Spring [2014]. Visited 2014-05-01. Licensed under the Creative Commons CC0 1.0 Universal Public Domain Dedication.
URL: *http://commons.wikimedia.org/w/index.php?title=File:Tuzla_unrest_2014-02-07_file_3.JPG&oldid=121745225*

[25] *Optical character recognition.* A Dictionary of Media and Communication. Oxford University Press, 1st ed. [2011]. ISBN 9780199568758.

[26] Rice, S.V., Nagy, G. and Nartker, T.A. *Optical Character Recognition: An Illustrated Guide to the Frontier*, vol. 502 of *The Kluwer international series in engineering and computer science.* Kluwer Academic Publishers [1999].

[27] Smith, R.W. History of the tesseract ocr engine: what worked and what didn't [2013]. doi:10.1117/12.2010051.
URL: *http://dx.doi.org/10.1117/12.2010051*

[28] Smith, R. An Overview of the Tesseract OCR Engine [2007]. Visited 2014-04-24.
URL: *http://tesseract-ocr.googlecode.com/svn/trunk/doc/tesseracticdar2007.pdf*

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

[30] Börjesson, E. and Feldt, R. Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry. In *IEEE Fifth International Conference on Software Testing, Verification and Validation.* ISBN 1457719061 [2012] pp. 350 – 359.

[31] Alégroth, E., Feldt, R. and Olsson, H.H. Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study. In *IEEE Sixth International Conference on Software Testing, Verification and Validation.* ISBN 9781467359610 [2013] pp. 56 – 65.

[32] Alégroth, E. Industrial Applicability of Visual GUI testing for System and Acceptance Test Automation. In *IEEE Fifth International Conference on Software Testing, Verification and Validation.* ISBN 1457719061 [2012] pp. 475 – 478.

[33] Levine, G. and DeJong, G. Object Detection by Estimating and Combining High-Level Features. In P. Foggia, C. Sansone and M. Vento, eds., *Image Analysis and Processing – ICIAP 2009*, vol. 5716 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg. ISBN 978-3-642-04145-7 [2009] pp. 161–169. doi:10.1007/978-3-642-04146-4_19. Visited: 2014-06-10.
URL: *http://dx.doi.org/10.1007/978-3-642-04146-4_19*

[34] opencv dev team. ml. Machine Learning – OpenCV 2.4.9.0 documentation [2014]. Visited: 2014-06-10.
URL: *http://docs.opencv.org/modules/ml/doc/ml.html*

[35] Saab AB. GIRAFFE AMB Multi-Role 3D Surveillance Radar GBAD [2014]. Visited: 2014-09-07.
URL: *http://www.saabgroup.com/en/Land/Ground_Based_Air_Defence/Ground-Based-Surveillance/Giraffe-AMB/*

[36] Alégroth, E., Nass, M. and Olsson, H.H. JAutomate: a Tool for System- and Acceptance-test Automation. In *IEEE Sixth International Conference on Software Testing, Verification and Validation.* ISBN 9781467359610 [2013] pp. 439 – 446.

[37] User Interface Design Group at MIT. Sikuli Script - Home []. Visited: 2014-03-10.
URL: *http://www.sikuli.org*

[38] Yeh, T., Chang, T.H. and Miller, R. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology.* ACM. ISBN 1605587451 [2009] pp. 183 – 192.

[39] Swifting AB. Features | JAutomate [2014]. Visited: 2014-03-10.
URL: *http://jautomate.com/features/*

[40] Inceptive AB. JAutomate - Tool for test automation - Inceptive (in Swedish) [2014]. Visited: 2014-03-10.
URL: *http://www.inceptive.se/jautomate/*

[41] Hocke, R. Sikuli Remote 2014 [2014]. Visited: 2014-07-01.
URL: *https://github.com/RaiMan/SikuliX-2014/tree/93936718b32598c755ec88dadd910565e2ad415e/Remote*

[42] Mishra, N., Patvardhan, C., Lakshmi, C.V. and Singh, S. Shirorekha chopping integrated tesseract ocr engine for enhanced hindi language recognition. *International Journal of Computer Applications* [2012] vol. 39(6).
URL: *http://search.proquest.com/docview/926414180*

[43] zdenop. Qt-box-editor []. Visited 2014-09-13.
URL: *https://zdenop.github.io/qt-box-editor/*

[44] Cheyne, J.A., Solman, G.J., Carriere, J.S. and Smilek, D. Anatomy of an error: A bidirectional state model of task engagement/disengagement and attention-related errors. *Cognition* [2009] vol. 111(1):pp. 98 – 113. ISSN 0010-0277. doi:http://dx.doi.org/10.1016/j.cognition.2008.12.009. Visited 2014-04-24.
URL: *http://www.sciencedirect.com/science/article/pii/S001002770900002X*

[45] Smallwood, J. and Schooler, J.W. The restless mind. *Psychological Bulletin* [2006] vol. 132(6):pp. 946–958.

[46] Hole, G. The Wilcoxon test [2011]. Visited: 2014-09-07.
URL: *http://sussex.ac.uk/Users/grahamh/RM1web/WilcoxonHandoout2011.pdf*

A

Training process for Tesseract

```
tesseract eng.image.exp0.png eng.image.exp0 batch.nochop makebox
Edit generated boxfile such that the found characters are correct
tesseract eng.image.exp0.png eng.image.exp0.box nobatch box.train
unicharset_extractor eng.image.exp0.box
echo "image.exp0 1 0 0 0 0" > font_properties
shapeclustering -F font_properties -U unicharset -O eng.unicharset
      eng.image.exp0.box.tr
mftraining -F font_properties -U unicharset -O eng.unicharset
      eng.image.exp0.box.tr
cntraining eng.image.exp0.box.tr
copy unicharset test.unicharset
copy normproto test.normproto
copy inttemp test.inttemp
copy pffmtable test.pffmtable
copy shapetable test.shapetable
combine_tessdata test.
```

Table A.1: Procedure for training Tesseract.