Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry

Emil Börjesson and Robert Feldt
Software Engineering and Technology
Chalmers University
Gothenburg, Sweden
emil.borjesson@chalmers.se

***Abstract*—Software companies are under continuous pressure to shorten time to market, raise quality and lower costs. More automated system testing could be instrumental in achieving these goals and in recent years testing tools have been developed to automate the interaction with software systems at the GUI level. However, there is a lack of knowledge on the usability and applicability of these tools in an industrial setting. This study evaluates two tools for automated visual GUI testing on a real-world, safety-critical software system developed by the company Saab AB. The tools are compared based on their properties as well as how they support automation of system test cases that have previously been conducted manually. The time to develop and the size of the automated test cases as well as their execution times have been evaluated. Results show that there are only minor differences between the two tools, one commercial and one open-source, but, more importantly, that visual GUI testing is an applicable technology for automated system testing with effort gains over manual system test practices. The study results also indicate that the technology has benefits over alternative GUI testing techniques and that it can be used for automated acceptance testing. However, visual GUI testing still has challenges that must be addressed, in particular the script maintenance costs and how to support robust test execution.**

***Keywords*-Visual GUI testing; Empirical; Industrial Study; Tool Comparison;**

I. Introduction

Market trends with demands for faster time-to-market and higher quality software continue to pose challenges for software companies that often work with manual test practices that can not keep up with increasing market demands. Companies are also challenged by their own systems that are often Graphical User Interface (GUI) intensive and therefore complex and expensive to test [1], especially since software is prone to changing requirements, maintenance, refactoring, etc., which requires extensive regression testing. Regression testing should be conducted with configurable frequency [2], e.g. after system modification or before software release, on all levels of a system, from unit tests, on small components, to system and acceptance tests, with complex end user scenario input data [3], [4]. However, due to the market imposed time constraints many companies are compelled to focus or limit their manual regression testing with ad hoc test case selection techniques [5] that do not guarantee testing of all modified parts of a system and cause faults to slip through.

Automated testing has been proposed as one solution to the problems with manual regression testing since automated tests can run faster and more often, decreasing the need for test case selection and thereby raising quality, while reducing manual effort. However, most automated test techniques, e.g. unit testing [6], [7], Behavioral Driven Development [8], etc., approach testing on a lower system level that has spurred an ongoing discussion regarding if these techniques, with certainty, can be applied on high-level system tests, e.g. system tests [9], [10]. This uncertainty has resulted in the development of automated test techniques explicit for system and acceptance tests, e.g. Record and Replay (R&R) [11]–[13]. R&R is a tool-supported technique where user interaction with a System Under Test's (SUT) GUI components are captured in a script that can later be replayed automatically. User interaction is captured either on a GUI component level, e.g. via direct references to the GUI components, or on a GUI bitmap level, with coordinates to the location of the component on the SUT's GUI. The limitation with this technique is that the scripts are fragile to GUI component change [14], e.g. API, code, or GUI layout change, which in the worst case can render entire automated test suites inept [15]. Hence, the state-of-practice automated test techniques suffer from limitations and there is a need for a more robust technique for automation of system and acceptance tests.

In this paper, we investigate a novel automated testing technique, which we in the following call visual GUI testing, with characteristics that could lead to more robust system test automation [16]. Visual GUI testing is a script based testing technique that is similar to R&R but uses image recognition, instead of GUI component code or coordinates, to find and interact with GUI bitmap components, e.g. images and buttons, in the SUT's GUI. GUI bitmap interaction based on image recognition allows visual GUI testing to mimic user behavior, treat the SUT as a black box, whilst being more robust to GUI layout change. It is therefore a prime candidate for better system and acceptance test automation. However, the body of knowledge regarding visual GUI testing is small and contain no industrial experience reports or other studies to support the techniques industrial applicability. Realistic evaluation on industrial scale testing problems are key in understanding and refining this technique. The body of knowledge neither

contains studies that compare different visual GUI testing tools or the strengths and weaknesses of the technique in the industrial context.

This paper aims to fill these gaps of knowledge by presenting a comparison of two visual GUI testing tools, one commercial referred to as CommercialTool[1], and one open source, called Sikuli [16], in an industrial context to answer the following research questions:

1) Is visual GUI testing applicable in an industrial context to automate manual high-level system regression tests?
2) What are the advantages and disadvantages of visual GUI testing for system regression testing?

To answer these questions we have conducted an empirical, multi-step case study at a Swedish company developing safety-critical software systems, Saab AB. A preparation step evaluated key characteristics of the two tools and what could be the key obstacles to applying it at the company. Dynamic evaluation of the tools was then done in an experimental setup to ensure the tools could handle key aspects of the type of system testing done at the company. Finally, a representative selection of system test cases for one of the company's safety-critical subsystems was automated in parallel with both of the tools. Our results and lessons learned give important insight on the applicability of visual GUI testing.

The paper is structured as follows; section II presents related work followed by section III that describes the case study design. Section IV presents results which are then discussed in section V. Section VI concludes the paper.

II. Related Work

The body of knowledge on using GUI interaction and image recognition for automation is quite large and has existed since the early 90s, e.g. Potter [17] and his tool Triggers used for GUI interactive computer macro development. Other early works includes Zettlemoyer and Amant who explored GUI automation with image recognition in their tool, VisMap. VisMap's capabilities were demonstrated through automation of a visual scripting program and the game Solitaire [18]. These early works did however not focus on automated testing but rather automation in general with the help of image recognition algorithms.

There is also a large body of knowledge on using GUI interaction for software testing, as shown by Adamoli et al. [11] who have surveyed 50 papers related to automated GUI testing for their work on GUI performance testing. Note that we differentiate between GUI interaction for automation and GUI interaction for testing since all techniques for GUI automation are not intended for testing and vice versa.

One of the most common GUI testing approaches is Record and Replay (R&R) [11]–[13]. R&R is based on a two step process where user mouse and keyboard inputs are first recorded and automatically stored in a script that the tool can then replay in the second step. Different R&R tools record user input on different GUI abstraction levels, e.g. the GUI object level or the GUI bitmap level, with different advantages and disadvantages for each level. On the top GUI bitmap level a common approach is to save the coordinates of the GUI interaction in a script, with the drawback that the script becomes sensitive to reconfiguration of GUI layout but with the advantage of making the scripts robust to API and code changes. The other R&R approach is to record SUT interaction on a lower GUI object level by saving references to the GUI code components, e.g. Java Swing components, which instead make the scripts sensitive to API and code structure change [15] but more robust to GUI layout reconfiguration.

GUI testing can also be conducted on the top GUI bitmap level with techniques that use image recognition to execute test scenarios [16], in this paper referred to as visual GUI testing. Visual GUI testing is very similar to the R&R approach but with the important distinction that R&R tools do not use image recognition and are thus more hardcoded to the exact positioning of GUI elements. In current visual GUI testing tools, the common approach is that scenarios are written manually in scripts that include images for SUT interaction in contrast to the R&R approach where test scripts are commonly generated automatically with coordinates or GUI component references. In a typical visual GUI testing script input is given to the SUT through automated mouse and keyboard commands to GUI bitmap components identified through image recognition, output is then observed, once again with image recognition, and compared to expected results after which the next sequence of input is given to the SUT, etc. The advantages of visual GUI testing is that it is impervious to GUI layout reconfiguration, API and code changes, etc., but with the disadvantage that it is instead sensitive to changes to GUI bitmap objects, e.g. change of image size, shape or color.

A different approach to GUI testing is to base it on models, e.g. generate test cases from finite state machines (FSM) [19], [20]. However, the models often need to be created manually at considerable cost and the approach often face scalability problems. Automated model creation approaches have been proposed, such as GUI ripping proposed by Memon [21].

Hence, the area of GUI interaction, automation and testing, is quite broad but limited regarding empirical studies evaluating the techniques on real-world, industrial-scale software systems. Comparative research has been done on tools that use the R&R technique [11], but, to the authors' knowledge, there are no studies that compare visual GUI testing tools or evaluate if they can substitute manual regression testing in the industrial context.

Another important test aspect is acceptance testing where user and customer requirement conformity is verified with test scenarios that emulate end user interaction with the SUT. The tests are similar to system test cases, but contain more end user specific interaction information, i.e. how the system will be used in its intended domain. Acceptance test scenarios should preferably be automated and run regularly to verify system conformity to the system requirements [2] and has therefore been subject to academic research. The academic research has resulted in both tools and frameworks for acceptance test

[1]For reasons of confidentiality we cannot disclose the name of the tool.

automation, including tools for GUI-interaction [22], but to the authors' knowledge there is no research using visual GUI testing for acceptance testing.

III. Case Study Description

The empirical study presented in this paper was conducted in a real-world, industrial context, in one business area of the company Saab AB, in the continuation of this paper referred to as Saab. Saab develops safety critical air traffic control systems that consist of several individual subsystems of which a key one was chosen as the subject of this study. The subsystem has in the order of 100K Lines of Code (LOC), constituting roughly one third of the functionality of the system it is part of, and is tested with different system level tests, including 50 manual scenario based system test cases. At the time of the study the subsystem was in the final phase for a new customer release that was one reason why it was chosen. Other reasons for the choice included the subsystem size in LOC, the number of manual test cases, and because it had a non-animated GUI. With non-animated we mean that there are no moving graphical components, only components that, when interacted with, change face, e.g. color. Decision support information for what subsystem to include in the study was gathered through document analysis, interviews and discussions with different software development roles at Saab.

CommercialTool was selected for this study because Saab had been contacted by the tool's vendor and been provided with a trial license for the tool that made it accessible. It is a mature product for visual GUI testing having been on the market since more than 5 years. The second tool, Sikuli, was chosen since it seemed to have similar functionality as CommercialTool and, if applicable, would be easier to refine and adapt further to the company context. The company was also interested in the relative cost benefits of the tools, i.e. if the functionality or support of CommercialTool would justify its increased up-front cost.

The methodology used in the study was divided into two main phases, shown in Figure 1, with three steps in each phase. Phase one of the study was a pre-study with three different steps. An initial tool analysis compared the tools based on their static properties as evaluated through ad hoc script development and review of the tools' documentation. This was followed by a series of experiments with the goal of collecting quantitative metrics on the strengths and weaknesses of the tools. The experiments also served to provide information about visual GUI testing's applicability for different types of GUIs, e.g. animated with moving objects and non-animated with static buttons and images, which would provide decision support for, and possibly rule out, what type of system to study at Saab in the second phase of the study. In parallel with these experiments an analysis of the industrial context at Saab was also conducted. Phase two of the study was conducted at Saab and started with a complete manual system test of all the 50 test cases of the studied subsystem. This took 40 hours, spread over five days, during which the manual test cases were categorized based on their level of possible automation with

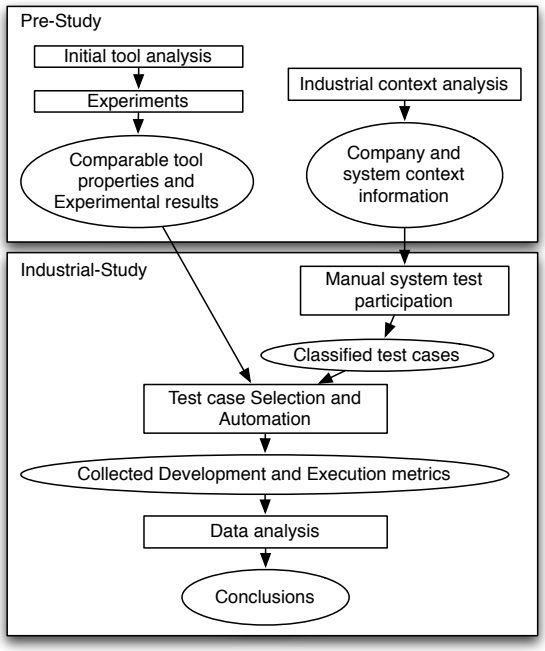

Fig. 1. Overview of research methodology (square nodes show activities/steps and rounded ones outcomes).

the visual GUI testing tools. Both of the visual GUI testing tools were then used to automate five, carefully selected, representative, test case scenarios (ten percent) of the manual test suite during which metrics on script development time, script LOC and script execution time were collected.

In the following sections the two phases of the methodology will be described in more detail.

A. Pre-study

Knowledge about the industrial context at Saab was acquired through document analysis, interviews and discussions with different roles at the company. The company's support made it possible to identify a suitable subsystem for the study, based on subsystem size, number of manual test cases, GUI properties, criticality, etc., and to identify the manual test practices conducted at the company.

In parallel with the industrial context analysis, static properties of the studied tools were collected, through explorative literature review of the tools' documentation and ad hoc script development. The collected properties were then analyzed according to the quality criteria proposed by Illes et al. [23], derived from the ISO/IEC 9126 standard supplemented with criteria to define tool vendor qualifications. The criteria refer to tool quality and are defined as *Functionality, Reliability, Usability, Efficiency, Maintainability, Portability, General vendor qualifications, Vendor support, and Licensing and pricing.*

The tools were also analyzed in four structured experiments where scripts were written in both tools, with equivalent instructions to make the scripts comparable, and then executed against controlled GUI input. The GUI input was classified into two groups, animated GUIs and non-animated GUIs, cho-

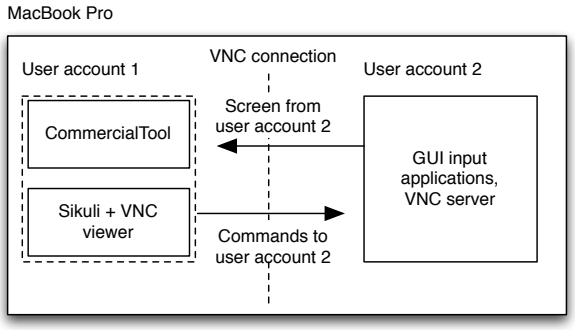

Fig. 2. Visualization of the experimental setup.

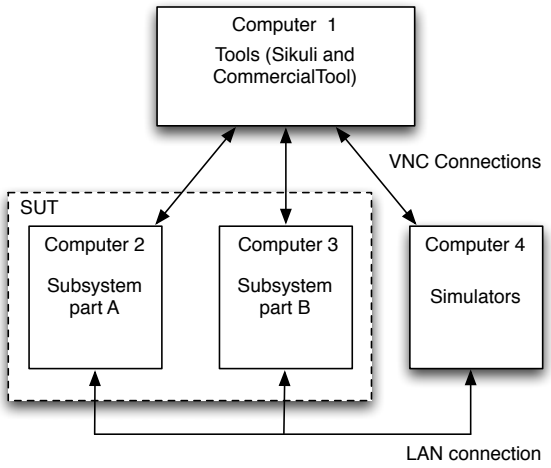

Fig. 3. Visualization of the test system setup.

sen to cover and evaluate how the tools perceivably performed for different types of industrial systems. The ability to handle animated GUIs is critical for visual GUI testing tools since they apply compute-intensive image recognition algorithms that might not be able to cope with highly dynamic GUIs. Eight scripts were written in total, four in each tool, and each one was executed in 30 runs for each experiment. The experiments have been summarized in the following list:

- Experiment 1: Aimed to determine how well the tools could differentiate between alpha-numerical symbols by adding the numbers six and nine in a non-animated desktop calculator by locating and clicking on the calculator's buttons.
- Experiment 2: Aimed to determine how the tools could handle small graphical changes on a large surface, tested by repeated search of the computer desktop for a specific icon to appear that was controlled by the researcher.
- Experiment 3: Aimed to test the tools image recognition algorithms in an animated context by locating the back fender of a car driving down a street in a video clip in which the sought target image was only visible for a few video frames.
- Experiment 4: Also in an animated context, aimed to identify how well the tools could track a moving object over a multi-colored surface in a video clip of an aircraft, represented by its textual call-sign, moving across a radar screen.

The four experiments cover typical functionality and behavior of most software system GUIs, e.g. interaction with static objects such as buttons or images, timed events and objects in motion, to provide a broad view of the applicability of the tools for different systems. Experiment 4 was selected since it is similar to one of the systems developed by the company.

The experiments were run on a MacBook Pro computer, with a 2.8GHz Intel Core 2 Duo processor, using virtual network computing (VNC) [24], which was a requirement for CommercialTool. CommercialTool is designed to be non-intrusive, meaning that it should not affect the performance of the SUT, and to support testing of distributed software systems. This is achieved by performing all testing over VNC and support for it is built into the tool. Sikuli does not have VNC support so to equalize the experiment conditions Sikuli was paired with a third party VNC viewer application. The VNC viewer application was run on one user account connected to a VNC server on a second user account on the experiment computer, visualized in Figure 2.

Finally the visual GUI testing tools were also analyzed in terms of learnability since this aspect affects the technique's acceptance, e.g. if the tool has a steep learning curve it is less likely to be accepted by users [25]. The learnability was evaluated in two ad hoc experiments using Sikuli, where two individuals with novice programming knowledge, at two different occasions, had to automate a simple computer desktop task with the tool.

B. Industrial Study

The studied subsystem at Saab consisted of two computers with the Windows XP operating system, connected through a local area network (LAN). The LAN also included a third computer running simulators, used during manual testing to emulate domain hardware controlled by the subsystem's GUI. The GUI consisted primarily of custom-developed GUI components, such as buttons and other bitmap graphics, and was non-animated. During the study a fourth computer was also added to the LAN to run the visual GUI testing tools and VNC, visualized in Figure 3. VNC is scalable for distributed systems so the level of complexity of the industrial test system setup, Figure 3, was directly comparable to the complexity of the experimental setup used during the pre-study, Figure 2.

In the first step of the industrial study the researchers conducted a complete manual system test of the chosen subsystem with two goals. The first goal was to categorize the manual test cases as fully scriptable, partially scriptable or not scriptable based on the tool properties collected during the pre-study. The categorization provided input for the selection of representative manual test cases to automate and showed if enough of the manual test suite could be automated for the automation to be valuable for Saab.

All the subsystem's manual test cases were scenario based, written in natural language, including pre- and post-conditions

for each test case and were organized in tables with three columns. Column one described what input to manually give to the subsystem, e.g. click on button x, set property y, etc. Column two described the expected result of the input, e.g. button x changes face, property y is observed on object z, etc. The last column was a check box where the tester should report if the expected result was observed or not. The test case table rows described the test scenario steps, e.g. after giving input x, observing output y and documenting the result in the checkbox on row k the scenario proceeded on row k+1, etc., until reaching the final result checkbox on row n. Hence, the test scenarios were well defined and documented in a way suitable as input for the automation.

The second research purpose of conducting the manual system test was to acquire information of how the different parts of the subsystem worked together and what or which test cases provided test coverage for which part(s) of the subsystem. Test coverage information was vital in the manual test case selection process to ensure that the selected test cases were representative for the entire test suite so that the results could be generalized. Generalization of the results was required since it was not feasible to automate all 50 of the subsystem's manual test cases during the study.

Five test cases were selected for automation with the goal of capturing as many mutually exclusive GUI interaction types as possible, e.g. clicks, sequences of clicks, etc., to ensure that these GUI interaction types, and in turn test cases including these GUI interaction types, could be automated. GUI interaction types with properties that added complexity to the automation were especially important to cover in the five automated test cases, the most complex properties have been listed below:

1) The number of physical computers in the subsystem the test case required access to.
2) Which of the available simulators for the subsystem the test case required access to.
3) The number of run-time reconfigurations of the subsystem the test case included.

The number of physical computers would impose complexity by requiring additional VNC control code and interaction with a broader variety of GUI components, e.g. interaction with custom GUI components in subsystem part A and B and the simulators. Simulator interaction was also important to cover in the automated test cases since if some simulator interaction could not be automated neither could the manual test cases using that simulator. Run-time reconfiguration in turn added complexity by requiring the scripts to read and write to XML files. In Table I the five chosen test cases have been summarized together with which of the three properties they automate. The minimum number of physical computers required in any test case were two and maximum three whilst the maximum number of run-time configurations in any test case were also three. There were four simulators, referred to as A, B, C and D, but only simulators A and B were automated in any script because they were the most commonly used in the manual test cases and also had the most complex GUIs. In addition, simulators C and D had very similar functionality to A and B and had no unique GUI components not present in A or B and were therefore identified as less important and possible to automate.

Test case	**Physical computers**	**Run-time config.**	**Simulator**
Test case 1	2	3	A
Test case 2	2	0	B
Test case 3	2	2	A
Test case 4	2	0	A
Test case 5	3	0	A

TABLE I
PROPERTIES OF THE MANUAL TEST CASES SELECTED FOR AUTOMATION. THE NUMBER OF PHYSICAL COMPUTERS DOES NOT INCLUDE THE COMPUTER USED TO RUN THE VISUAL GUI TESTING TOOLS.

Once the representative test cases had been selected from the manual test suite they were automated in both of the studied tools during which metrics were collected for comparison of the tools and the resulting scripts. Metrics that were collected included script development time, script LOC and script execution time.

IV. RESULTS

Below the results gathered during the study are presented divided into the results gathered during the pre-study and the results gathered during the industrial phase of the study.

A. Results of the Pre-study

The pre-study started with a review of the studied visual GUI testing tools' documentation from which 12 comparable static tool properties relevant for Saab were collected. The 12 properties are summarized in Table II that shows which property had impact on what tool quality criteria defined by Illes et al. [23], described in section III. The table also shows what tool was the most favorable to Saab in terms of a given property, e.g. CommercialTool was more favorable in terms of real-time feedback than Sikuli. The favored tool is represented in the table with an S for Sikuli, CT for CommercialTool and (-) if the tools were equally favorable.

In the following section each of the 12 tool properties are discussed in more detail, compared between the tools and related to what tool quality criteria they impact.

Developed in. CommericalTool is developed in C#, whilst Sikuli is developed in Jython (a Python version in Java), which is relevant for the portability of the tools since CommercialTool only works on certain software platforms whilst Sikuli is platform independent. Sikuli, being open source, also allows the user to expand the tool with new functionality, written in Jython, whilst users of CommercialTool must rely on vendor support to add tool functionality.

Script Language syntax. The script language in Sikuli is based on Python, extended with functions specific for GUI interaction, e.g. clicking on GUI objects, writing text in a GUI, waiting for GUI objects, etc. Sikuli scripts are written in the tool's Integrated Development Environment

(IDE) and because of the commonality between Python and other imperative/Object-Oriented languages the tool has both high usability and learnability with perceived positive impact on script maintainability. The learnability of Sikuli is also supported by the learnability experiments conducted during the pre-study, described in Section III, where novice programmers were able to develop simple Sikuli scripts after only 10 minutes of Sikuli experience and advanced scripts after an hour.

CommercialTool has a custom scripting language, modelled to resemble natural language that the user writes in the tool's IDE, which has a lot of functionality, but the tool's custom language has a higher learning curve than Sikuli script. The usability of CommercialTool is however strengthened by the script language instruction-set that is more extensive than the instruction-set in Sikuli, e.g. including functionality to analyze audio output, etc. Both Sikuli and CommercialTool do however support all the most common GUI interaction functions and programming constructs, e.g. loops, switch statements, exception handling, etc.

Supports imports. Additional functionality can be added to Sikuli by user-defined imports written in either Java or Python code to extend the tool's usability and efficiency. CommercialTool does not support user-defined imports and again users must rely on vendor support to add tool functionality.

Image representation in tool IDE. Scripts in CommercialTool refers to GUI interaction objects (such as images) through textual names whilst Sikuli's IDE shows the GUI interaction objects as images in the script itself. The image presentation in Sikuli's IDE makes Sikuli scripts very intuitive to understand, also for non-developers, which positively affects the usability, maintainability and portability of the scripts between versions of a system. In particular this makes a difference for large scripts with many images.

Real-time script execution feedback. CommercialTool provides the user with real-time feedback, e.g. what function of the script is currently being executed and success or failure of the script. Sikuli on the other hand executes the script and then presents the user with feedback, i.e. post script execution feedback. This lowers the usability and maintainability of test suites in Sikuli since it becomes harder to identify faults.

Image recognition sweeps per second. Sikuli has one image recognition algorithm that can be run five times every second whilst the image recognition algorithm in CommercialTool runs seven times every second. CommercialTool is therefore potentially more robust, e.g. to GUI timing constraints, and have higher reliability and usability, at least in theory, than Sikuli for this property.

Image recognition failure mitigation. CommercialTool has several image recognition algorithms with different search criteria that give the tool higher reliability, usability, efficiency, maintainability and portability by providing automatic script failure mitigation. Script failure mitigation in Sikuli requires manual effort, e.g. by additional failure mitigation code or by setting the similarity, 1 to 100 percent, of a bitmap interaction object required for the image recognition algorithm to find a match in the GUI. Hence, Sikuli has less failure mitigation functionality that can have negative effects on usability, reliability, etc.

Property	**CommercialTool**	**Sikuli**	**Impacts**	**Favored tool**
Developed in	C#	Jython	F/P/VS	S
Script language syntax	Custom	Python	F/U/M	S
Supports imports	No	Java and Python	F/U/E/VS	S
Image representation in tool IDE	Text-Strings	Images	F/U/M/P	S
Real-time script execution feedback	Yes	No	U/M	CT
Image recognition sweeps per second	7	5	F/R/U	CT
Image recognition failure mitigation	Multiple algorithms to choose from	Image similarity configuration	F/R/U/E/M/P	CT
Test suite support	Yes	Unit tests only	F/U/M/P	-
Remote SUT connection support	Yes	No	F/U/P	-
Remote SUT connection requirement	Yes	No	F/U/P	S
Cost	10.000 Euros per license per computer	Free	U/LP	S
Backwards compatibility	Guaranteed	Uncertain	F/M/GVQ	CT

TABLE II
RESULTS OF THE PROPERTY COMPARISON BETWEEN COMMERCIALTOOL AND SIKULI. COLUMN **IMPACTS**: F - FUNCTIONALITY, R - RELIABILITY, U - USABILITY, E - EFFICIENCY, M - MAINTAINABILITY, P - PORTABILITY, GVQ - GENERAL VENDOR QUALIFICATIONS, VS - VENDOR SUPPORT, LP - LICENSING AND PRICING. COLUMN **FAVORED TOOL**: S - SIKULI, CT - COMMERCIALTOOL, **(-)** - EQUAL BETWEEN THE TOOLS

Test suite support. Sikuli does not have built in support to create, execute or maintain test suites with several test scripts, only single unit tests. CommercialTool has such support built in. A custom test suite solution was therefore developed during the study that uses Sikuli's import ability to run several test scripts in sequence, providing Sikuli with the same functionality, usability, perceived maintainability and portability.

Remote SUT connection support / requirement. Sikuli does not have built in VNC support, a property that is not only supported by CommercialTool but also required by the tool to operate. Sikuli was therefore paired with a third party VNC application as described in Section III, to provide Sikuli with the same functionality, usability and portability as CommercialTool.

Cost. The studied tools differ in terms of cost since Sikuli is open source with no up-front cost whilst CommercialTool costs around 10.000 Euros per 'floating license' per year. A floating license means that it is not connected to any one user or computer but only one user can use the tool at a time, hence the Licensing and pricing quality criterion in this case affects the usability of CommericalTool since some companies may

Experiment	Type	Desc.	CT success rate (%)	Sikuli success rate (%)
1	non-animated	Calculator	100	50
2	non-animated	Icon finder	100	100
3	animated	Car Finder	3	25
4	animated	Radar trace	0	100

TABLE III
ACADEMIC EXPERIMENT RESULTS. CT STANDS FOR COMMERCIALTOOL. TYPE INDICATES IF THE EXPERIMENT WAS NON-ANIMATED OR NOT AND DESC. DESCRIBES THE EXPERIMENT.

not afford multiple licenses while still wanting to run multiple scripts at the same time.

Backwards compatibility and support. The last property concerns the backwards compatibility of the tools, and whilst CommercialTool's vendor guarantees that the tool, which has been available in market for several years, will always be backwards compatible, Sikuli is still in beta testing and therefore subject to change. Changes to Sikuli's instruction set could affect the functionality and maintainability of the tool and scripts. This property also provides general vendor qualification information, e.g. the maturity of the vendor and the tool, which plays an important part for tool selection and tool adoption in a company, e.g. that CommercialTool may be favored because it is more mature and the tool vendor can supply support etc.

The second part of the pre-study consisted of four structured experiments, described in Section III and their results are summarized in Table III. In the first experiment a script was developed in each tool for a non-animated desktop calculator application to evaluate CommercialTool's and Sikuli's image recognition algorithms' ability to identify alpha-numeric symbols. Sikuli only had a success rate of 50 percent in this experiment, over 30 runs, because the tool was not always able to distinguish between the number 6 and the letter C, used to clear the calculator, whilst CommercialTool had a success rate of 100 percent. In the second experiment the goal was to find a specific icon as it appeared on the desktop, hence identify a small bitmap change on a large surface, for which both tools had a 100 percent success rate. In the third experiment the goal was to identify the back fender of a car driving down a road in a video clip where the sought fender image was only visible for a few video frames, imposing a time constraint to the image recognition algorithms. The car experiment resulted in Sikuli having a success rate of 25 percent and CommercialTool 3 percent. The final experiment required the tools to trace the call sign, a text string, of an aircraft moving over a multi-colored radar screen in a video-clip, where Sikuli had a 100 percent success rate whilst CommercialTool's success rate was 0 percent.

A summary of the pre-study results show that CommercialTool had higher success rate in the experiments with non-animated GUIs and had more built-in functionality required for automated testing in the industrial context, shown by the 12 analyzed properties. Sikuli on the other hand had higher success rate in the experiments with animated GUIs and showed to be easier to adapt, only requiring small efforts to be extended with additional functionality. In addition, Sikuli was considered marginally favored according to the tool quality criteria defined by Illes et al. and is therefore perceived as a better candidate for future research.

B. Results of the industrial study

The industrial part of the study started with the researchers conducting a complete manual system test of the studied subsystem. During the manual system test all the test cases were analyzed, as described in Section III, and classified into categories. The category analysis showed that Sikuli could fully script 81%, partially script 17% and not script 2% of the manual test cases. CommercialTool on the other hand could fully script 95%, partially script 3% and not script 2% of the manual test cases. The higher percentage of scripts that could be fully automated in CommercialTool was given by the tool's ability to analyze audio output, required in seven of the manual test cases. The 2% of the manual test cases that could not be scripted, in either tool, were hardware related and required physical interaction with the SUT.

Based on the categorization and the selection criteria, discussed in Section III, five manual test cases were chosen for automation. The automation was done pair-wise in each tool, e.g. test case x was automated in one tool and then in the other tool, with the order of the first tool chosen at random for each test case. Random tool selection was used to ensure that the script development time for the script developed in the secondly used tool would not continuously be skewed, lowered, because challenges with the script, e.g. required failure mitigation, etc., had already been resolved when the script was developed in the first tool.

The main contributor to script development time was in the study observed to be the amount of code required to mitigate failure due to unexpected system behavior, e.g. GUI components not rendering properly, GUI components appearing on top of each other, etc. Failure mitigation was achieved through ad hoc addition of wait functions, conditional branches and other exception handling, e.g. try-catch blocks, which for each added function required extra image recognition sweeps of the GUI that also increased the script execution time. Scripts that required failure mitigation also took longer to develop since they had to be rerun more times during development to ensure script robustness. The development time required to make a script robust also proved to be very difficult to estimate because unexpected system behavior was almost never related to the test scenarios but rather a product of the subsystem's implementation. Each script was developed individually and consisted of three parts. First a setup part to cover the preconditions of the test case. The second part was the test scenario and the third part was a test teardown to put the subsystem back in a known state to prepare it for the following test case. After the five test scripts had been developed in each tool the LOC and execution time for each script was recorded, shown in Table IV together with the script

	CT			Sikuli			
Test case	**Dev-time (min)**	**Exe-time (sec)**	**LOC**	**Dev-time (min)**	**Exe-time (sec)**	**LOC**	**TC Steps**
ATC-1	255	111	103	105	90	212	5
ATC-2	195	405	233	200	390	228	4
ATC-3	285	390	368	260	338	345	16
ATC-4	205	80	80	180	110	92	9
ATC-5	120	90	115	150	154	169	8
Total:	17 hours 40 minutes	17.93 minutes	899 LOC	15 hours 55 minutes	18.00 minutes	1046 LOC	

TABLE IV
METRICS COLLECTED DURING TEST CASE AUTOMATION. CT STANDS FOR COMMERCIALTOOL, ATC FOR AUTOMATED TEST CASE AND TC STEPS FOR THE NUMBER OF TEST STEPS IN THE SCENARIO OF THE MANUAL TEST CASE.

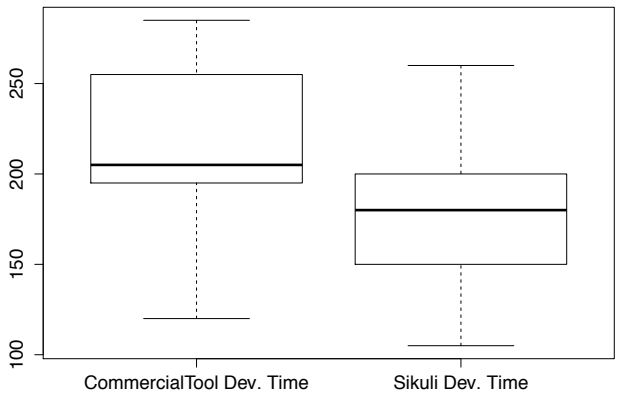

Fig. 4. Boxplot showing development time of the five scripts in each tool.

development time and number of steps in the corresponding manual test case scenario.

Table IV shows that the total development time, LOC and execution time were similar for the scripts in both tools.

The five chosen test cases were carefully selected to be representative for the entire manual test suite for the subsystem, as described in section III, to allow the collected data to be used for estimation. Estimation based on the average execution times, from Table IV, shows that the fully automated test suite for the subsystem, all 50 test cases, would run in approximately three and a half hours in each tool. A three and a half hour execution time constitutes a gain of 78 percent compared to the execution time of the current manual test suite, 16 hours, if conducted by an experienced tester. Hence, automation would constitute not only an advantage in that it can be run automatically without human input but a considerable gain in total execution time which allows for more frequent testing. Potentially tests can run every night and over weekends and shorten feedback cycles in development. In Figure 4 the script development time for the scripts, taken from Table IV, have been visualized in a box-plot that shows the time dispersion, mean development time, etc. Using the mean development time, the development time for the entire automated test suite, all 50 test cases, can be estimated to approximately 21 business days for CommercialTool and 18 business days for Sikuli. The estimated development time for the automated test suite is in the same order of time that Saab spends on testing during one development cycle of the subsystem. Hence, the investment of automating the test suite is perceived to be cost beneficial after one development cycle of the subsystem.

The data in Table IV was also subject to statistical tests to see if there was any statistical significant difference between the two tools. The data was first analyzed with a Shapiro-Wilks test of the difference between the paired variables in Table IV, which showed that the data was normally distributed. Normal distribution allowed the data to be analyzed further with the Student t-test that had the p-value results 0.3472 for development time, 0.956 for execution time and 0.2815 for LOC. The Student t-test results were then verified with a non-parametric paired Wilcoxon test that had results with the same statistical implications. Hence, both the Student t- and Wilcoxon-tests showed that we cannot reject the null hypothesis, H_0, on a 0.05 confidence level. Therefore, it can be concluded that there is no statistical significant difference between the scripts of the studied tools in terms of development time, execution time or LOC. The statistical results are however limited by the few data points the tests were conducted on.

V. DISCUSSION

Our study shows several differences between the two studied tools but that both tools were able to successfully automate 10 percent of an industrial manual system test suite, for which 98 percent of the test cases can be fully or partially automated with visual GUI testing. The open-source tool, Sikuli, had a higher percentage of test cases that could only be partially scripted since it has no current support for detecting audio output. However, this is not a major obstacle since either the audio output can be visualized, and thus tested visually, or Sikuli can be extended with Operating System (OS) system calls.

CommercialTool and Sikuli differ in terms of cost, vendor support, test functionality, script languages, etc., with impacts on different tool quality criteria, shown in Table II, and are all important properties to consider for the industrial applicability of visual GUI testing. However, to show that visual GUI testing has any applicability at all in industry the most important aspect concerns the functionality of the image recognition algorithms.

The image recognition algorithms are what sets visual GUI testing apart from other GUI testing techniques, e.g. R&R, and also determine for what types of systems it is possible to apply the technique. R&R that interacts through GUI components

was determined as unsuitable for the automation of the subsystem test cases since they had to interact with components not developed by Saab, e.g. interaction with custom and OS GUI components. These interactions required access to GUI component references that could not be acquired. The GUI components in the SUT, e.g. the simulators, windows in the OS, etc., did not always appear in the same place on the screen when launched. This behavior also ruled out R&R with coordinate interaction as an alternative for the study. Evaluation of visual GUI testing showed that it does not suffer from R&R's limitations and therefore works in contexts where R&R cannot be applied. Visual GUI testing is applicable on different types of GUIs, evaluated in the pre-study experiments and in industry, which showed that both studied tools had high success-rates with non-animated GUIs and that Sikuli also had good success-rate on animated GUIs as well. Hence, this study shows that visual GUI testing works for tests on non-animated GUIs and perceivably also for animated GUIs. Non-animated GUI applicability is however a subject for future deeper research.

The purpose of automation of manual tests is to make the regression testing more cost-efficient by increasing the execution speed and frequency and lower the required manual effort of executing the tests cases. Estimations based on the collected data show that a complete automatic test suite for the studied subsystem would execute in three and a half hours, which constitutes a 78 percent reduction compared to manual test execution with an experienced tester. Hence, the automated test suite could be run daily, eliminating the need for partial manual system tests, reduce cost, increase test frequency and lower the risk of slip through of faults. Mitigation of slip through of faults is however limited with this technique by the test scenarios since faulty functionality not covered by the test scripts would be overlooked, whilst a human tester could still detect them through visual inspection. Hence, the automated scripts cannot replace human testers and should rather be a complement to other test practices, such as manual free-testing. The benefit of visual GUI testing scripts compared to a human tester in terms of test execution is that the scripts are guaranteed to run according to the same sequence every time, whilst human testers are prone to take detours and make mistakes during testing, e.g. click on the wrong GUI object, etc., which can cause faults to slip through.

Scenario based system tests are very similar to acceptance tests and based on the results of this study it should therefore be concluded as plausible to automate acceptance tests with visual GUI testing. This conclusion is supported by the research of similar GUI testing techniques, e.g. R&R, which has been shown to work for acceptance test automation [12], [22]. Further support is provided by the fact that some of the manual test cases, categorized as fully scriptable, for the studied subsystem had been developed with customer specific data. The results of this study therefore provide initial support that visual GUI testing can be used for automated acceptance testing in industry.

During the study it was established that the primary cost of writing visual GUI testing scripts was related to the effort required to make the scripts robust to unexpected system behavior. Unexpected system behavior can be caused by faults in the system, related or unrelated to the script, and must be handled to avoid that these faults are overlooked or break the test execution. Other unexpected behavior can be caused by events triggered by the system's environment, e.g. warning messages displayed by the OS. Hence, events that may appear anywhere on the screen. These events can be handled with visual GUI testing but are a challenge for R&R since the events location, the coordinates, are usually nondeterministic. Script robustness in visual GUI testing can be achieved through ad hoc failure mitigation but is a time-consuming practice. A new approach, e.g. a framework or guidelines, is therefore required to make robust visual GUI test script development more efficient. Hence, another subject for future research.

The cost of automating the manual test suite for the studied subsystem was estimated to 20 business days, which is a considerable investment, and to ensure that it is cost-beneficial the maintenance costs of the suite therefore have to be small. Small is in this context measured compared to the cost of manual regression testing, hence the initial investment and the maintenance costs have to break even with the cost of the manual testing within a reasonable amount of time. The maintenance costs of visual GUI testing scripts when the system changes are however unknown and future research is needed.

Our results show that visual GUI testing is applicable for system regression testing of the type of industrial safety critical GUI based systems in use at Saab. The technique is however limited to find faults defined in the scripted scenarios. Hence, visual GUI testing cannot replace manual testing but minimize it for customer delivery. Visual GUI testing also allows tests to be run more often and are more flexible than other GUI testing techniques, e.g. coordinate based R&R, because of image recognition that can find a GUI component regardless of its position in the GUI. Furthermore, R&R tools that require access to the GUI components, in contrast to visual GUI testing, are not easily applicable at this company since their systems have custom-developed GUIs as required in their domain. We have also seen that visual GUI testing can be applied for automated acceptance testing. Being able to continuously test the system with user-supplied test data could have very positive results on quality.

Evaluating a technique's applicability in a real-world context is a complex task. We have opted on a multi-step case study that covers multiple different criteria that gives the company better decision support on which to proceed. Even though the test automation comparison is based on a limited number of test cases the research was designed so that these test cases are representative of the rest of the manual test suite. Still, this is a threat to the validity of our results. Our industrial partner is more concerned with the amount of maintenance that will be needed as the system evolves. If these costs are high they will seriously limit the long-term applicability of visual GUI testing.

VI. Conclusion

In this paper we have shown that visual GUI testing tools are applicable to automate system and acceptance tests for industrial systems with non-animated GUIs with both cost and potentially quality gains over state-of-practice manual testing. Experiments also showed that the open source tool that was evaluated can successfully interact with dynamically changing, animated GUIs that would broaden the number and type of systems it can be successfully applied to.

We present a comparative study of two visual GUI testing script tools, one commercial and one open source, at the company Saab AB. The study was conducted in multiple steps involving both static and dynamic evaluation of the tools. One of the company's safety critical subsystems, distributed over two physical computers, with a non-animated GUI, was chosen and 10 percent, 5 out of 50, representative, manual, scenario-based, test cases were automated in both tools. A pre-study helped select the relevant test cases to automate as well as evaluate the strengths and weaknesses of the two tools on key criteria relevant for the company.

Analysis of the tools properties show differences in the tools functionality but overall results show that both studied tools work equally well in the industrial context with no statistically significant differences in either development time, run time or LOC of the test scripts. Analysis of the subsystem test suite show that up to 98 percent of the test cases can be fully or partially automated using visual GUI testing with gains to both cost and quality of the testing. Execution times of the automated test cases are 78% lower than running the same test cases manually and the execution requires no manual input.

Our analysis shows that visual GUI testing can overcome the obstacles of other GUI testing techniques, e.g. Record and Replay (R&R). R&R either requires access to the code in order to interact with the System Under Test (SUT) or is tied to specific physical placement of GUI components on the display. Visual GUI testing is more flexible, interacting with GUI bitmap components through image recognition, and robust to changes and unexpected behavior during testing of the SUT. Both of these advantages were important in the investigated subsystem since it had custom GUI components and GUI components that changed position between test executions. However, more work is needed to extend the tools with ways to specify and handle unexpected system events in a robust manner; the potential for this in the technique is not currently well supported in the available tools. For testing of safety-critical software systems there is also a concern that the automated tools are not able to find defects that are outside the scope of the test scenarios, such as safety defects. Thus any automated system testing will still have to be combined with manual system testing before delivery but the main concern for future research is the maintenance costs of the scripts as a system evolves.

References

[1] P. Li, T. Huynh, M. Reformat, and J. Miller, "A practical approach to testing gui systems," *Empirical Software Engineering*, vol. 12, no. 4, pp. 331–357, 2007.

[2] R. Miller and C. Collins, "Acceptance testing," *Proc. XPUniverse*, 2001.

[3] P. Hsia, D. Kung, and C. Sell, "Software requirements and acceptance testing," *Annals of software Engineering*, vol. 3, no. 1, pp. 291–317, 1997.

[4] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen, "Behavior-based acceptance testing of software systems: a formal scenario approach," in *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*. IEEE, 1994, pp. 293–298.

[5] T. Graves, M. Harrold, J. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[6] M. Olan, "Unit testing: test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.

[7] E. Gamma and K. Beck, "Junit: A cook's tour," *Java Report*, vol. 4, no. 5, pp. 27–38, 1999.

[8] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North, "The rspec book: Behaviour driven development with rspec, cucumber, and friends," *Pragmatic Bookshelf*, 2010.

[9] E. Weyuker, "Testing component-based software: A cautionary tale," *Software, IEEE*, vol. 15, no. 5, pp. 54–59, 1998.

[10] S. Berner, R. Weber, and R. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 571–579.

[11] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated gui performance testing," *Software Quality Journal*, pp. 1–39, 2011.

[12] J. Andersson and G. Bache, "The video store revisited yet again: Adventures in gui acceptance testing," *Extreme Programming and Agile Processes in Software Engineering*, pp. 1–10, 2004.

[13] A. Memon, "Gui testing: Pitfalls and process," *IEEE Computer*, vol. 35, no. 8, pp. 87–88, 2002.

[14] M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth, "Automating performance testing of interactive java applications," in *Proceedings of the 5th Workshop on Automation of Software Test*. ACM, 2010, pp. 8–15.

[15] E. Sjösten-Andersson and L. Pareto, "Costs and benefits of structure-aware capture/replay tools," *SERPS'06*, p. 3, 2006.

[16] T. Chang, T. Yeh, and R. Miller, "Gui testing using computer vision," in *Proceedings of the 28th international conference on Human factors in computing systems*. ACM, 2010, pp. 1535–1544.

[17] R. Potter, *Triggers: Guiding automation with pixels to achieve data access*. University of Maryland, Center for Automation Research, Human/Computer Interaction Laboratory, 1992, pp. 361–382.

[18] L. Zettlemoyer and R. St Amant, "A visual medium for programmatic control of interactive applications," in *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*. ACM, 1999, pp. 199–206.

[19] A. Memon, M. Pollack, and M. Soffa, "Hierarchical gui test case generation using automated planning," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 144–155, 2001.

[20] P. Brooks and A. Memon, "Automated gui testing guided by usage profiles," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 333–342.

[21] A. Memon, "An event-flow model of gui-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[22] C. Lowell and J. Stell-Smith, "Successful automation of gui driven acceptance testing," *Extreme Programming and Agile Processes in Software Engineering*, pp. 1011–1012, 2003.

[23] T. Illes, A. Herrmann, B. Paech, and J. Rückert, "Criteria for software testing tool evaluation. a task oriented view," in *Proceedings of the 3rd World Congress for Software Quality*, vol. 2, 2005, pp. 213–222.

[24] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual network computing," *Internet Computing, IEEE*, vol. 2, no. 1, pp. 33–38, 1998.

[25] L. Fowler, J. Armarego, and M. Allen, "Case tools: Constructivism and its application to learning and usability of software engineering tools," *Computer Science Education*, vol. 11, no. 3, pp. 261–272, 2001.