On the Industrial Applicability of Visual GUI Testing

Emil Alégroth
On the Industrial Applicability of Visual GUI Testing

EMIL ALÉGROTH

Copyright ©2013 Emil Alégroth
except where otherwise stated.
All rights reserved.

Technical Report No 100L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden

This thesis has been prepared using \LaTeX. 
Printed by Chalmers Reproservice, 
Göteborg, Sweden 2013.
To Therese and my supporting family
Abstract

There is constant pressure on the software development industry to improve software quality and deliver new and innovative software faster and more efficiently. These demands affect all aspects of software development, from requirements engineering to testing. Today, much testing is performed with costly, tedious and error-prone manual practices, especially on higher levels of system abstraction such as in system and acceptance testing. Increased test automation has been proposed as a key solution to help alleviate these test-related problems and has found increasing use. Automated tests typically consist of scripts that can be executed to give frequent feedback on the system’s quality, ensure that previously correct functionality has not been negatively affected by changes, and also relieve resources e.g. human testers. However, as a system evolves, changes to the requirements also require maintenance of the tests, which can be costly. Furthermore, most automated test techniques approach testing from a lower level of system abstraction, e.g. testing the low-level components and functions, but their use for system testing has been questioned. Test techniques have been proposed that automate the interaction with the graphical user interface (GUI) of a system. For example, record and replay tools can be used to automate system testing by emulating a user interacting with the system. Existing literature have pointed out several shortcomings of these techniques, for example they are sensitive to GUI layout and code changes, require access to source code, are dependent on a specific platform or operating system access etc.

In this thesis we evaluate if Visual GUI Testing (VGT), a novel technique that combines test scripts with image recognition, can be used to automate high-level, software testing. The strength of VGT lies in the use of image recognition that makes the technique robust in instances where previous GUI-based test techniques had limitations. Yet, VGT is only sparsely applied in industry and the academic body of knowledge on the technique is limited; there is a lack of empirical evaluation and experience reports from industrial use.

To fill this gap in knowledge we have conducted three empirical case studies applying different VGT tools in industrial software projects at two Swedish companies. We have also conducted an industrial survey with participants from multiple companies. These studies provide evidence that VGT can be successfully applied in industrial software development projects, but also detail challenges, problems and limitations that need to be overcome for more widespread industrial adoption. Future work should focus on evaluating the long-term maintenance costs of VGT test scripts, since our studies only present initial evidence that these costs are manageable in relation to the benefits of the technique.

Keywords

Acknowledgment

First, my humblest thanks go out to my main supervisor, friend and mentor Professor Robert Feldt without whose unwavering support this thesis would never had been completed. You continuously push the bar higher for me, for which I am grateful, and I promise to never stop fighting until I clear it!

Secondly, my thanks go out to my second supervisor Associate professor Helena Holmström-Olsson, whose knowledge and joyful spirit always inspires me, both in times of joy and despair. I also would like to thank my examiner Professor Jörgen Hansson and my other current, and past, colleagues at the Software Engineering division at Chalmers University of Technology. Your help and guidance has been, and continues to be, invaluable. Special thanks go to Dr. Ana Magazinus, Ali Shahrokni, Joakim Pernstål and Pariya Kashfi for supporting me in my work, and for listening to my excessive ramblings about cars. I also want to thank Bogdan Marculescu and Professor Tony Gorschek, together with Robert, for influencing me to go into academia.

Another special person, to whom I am forever grateful, and whose support made this thesis possible is my loving wife Therese Alégroth. Without your help and love during the trials of my work I would not have made it. I would also especially like to thank my mother Anette, my father Tomas and my sister Mathilda Börjesson for their support and sacrifices to ensure that I could pursue this dream. Further I would like to thank my loving friends and family, perhaps one day you will understand what it is I do.

Furthermore I would like to thank my industrial contacts Per Lenberg and the staff at Saab AB and Michel Nass and the staff at Inceptive for their support during my research. I look forward to our continued collaboration and joint exploration of the many questions that still affect our research.

This research has been carried out in a joint research project financed by the Swedish Governmental Agency of Innovation Systems (VINNOVA), Chalmers University of Technology and Saab AB. My PhD studies was also been supported by the Swedish National Research School for Verification and Validation Excellence (SWELL) based on funding from Vinnova.
List of Publications

Appended papers

This thesis is based on the following papers:

[A] E. Börjesson, R. Feldt “Structuring Software Engineering Case Studies to Cover Multiple Perspectives”


Accepted to the 6th International Conference on Software Testing Verification and Validation (ICST’2013), Luxembourg, March 18-22, 2013.

In submission to the Empirical Software Engineering Journal.

Accepted to the 6th International Conference on Software Testing, Verification and Validation (ICST’2013), Luxembourg, March 18-22, 2013.
## Contents

Abstract v

Acknowledgment vii

List of Publications ix

1 Introduction 1
   1.1 Introduction .......................... 1
   1.2 Background and related work .......... 3
      1.2.1 System and Acceptance testing .... 5
      1.2.2 Automated Testing ................. 6
      1.2.3 Automated Graphical User Interface Testing .... 8
         1.2.3.1 Coordinate-based Record and Replay ... 8
         1.2.3.2 GUI component-based Record and Replay ... 9
      1.2.4 Visual GUI Testing ................. 11
   1.3 Research Focus ........................ 13
   1.4 Methodology .......................... 14
      1.4.1 Research Methodology .............. 14
      1.4.2 Research Validity and Reliability ... 15
      1.4.3 Experiments ........................ 15
      1.4.4 Interviews .......................... 17
      1.4.5 Surveys ............................. 18
      1.4.6 Workshops .......................... 19
      1.4.7 Other ............................... 19
   1.5 Chapter/Study Summaries .............. 20
      1.5.1 Chapter A: Measuring a research phenomena from multiple perspectives ... 20
      1.5.2 Chapter B: Visual GUI Testing in practice, initial support 22
      1.5.3 Chapter C: Visual GUI Testing in an industrial project 23
      1.5.4 Chapter D: The challenges, problems and limitations of Visual GUI Testing .... 24
      1.5.5 Chapter E: JAutomate, a tool for Visual GUI Testing 26
   1.6 Discussion ............................ 27
      1.6.1 Contributions ........................ 27
      1.6.2 Limitations .......................... 29
      1.6.3 Future Research ........................ 31
         1.6.3.1 The long-term feasibility of Visual GUI Testing 33
   1.7 Conclusions ........................... 34

xi
<table>
<thead>
<tr>
<th>2 Paper A</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>38</td>
</tr>
<tr>
<td>2.2 Research Context and the studied Company</td>
<td>39</td>
</tr>
<tr>
<td>2.3 The BAPO/PCF Framework</td>
<td>40</td>
</tr>
<tr>
<td>2.4 Discussion</td>
<td>46</td>
</tr>
<tr>
<td>2.5 Conclusions</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Paper B</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>50</td>
</tr>
<tr>
<td>3.2 Related Work</td>
<td>51</td>
</tr>
<tr>
<td>3.3 Case Study Description</td>
<td>53</td>
</tr>
<tr>
<td>3.3.1 Pre-study</td>
<td>54</td>
</tr>
<tr>
<td>3.3.2 Industrial Study</td>
<td>56</td>
</tr>
<tr>
<td>3.4 Results</td>
<td>58</td>
</tr>
<tr>
<td>3.4.1 Results of the Pre-study</td>
<td>58</td>
</tr>
<tr>
<td>3.4.2 Results of the industrial study</td>
<td>62</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>65</td>
</tr>
<tr>
<td>3.6 Conclusion</td>
<td>67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Paper C</th>
<th>69</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>70</td>
</tr>
<tr>
<td>4.2 Related Work</td>
<td>71</td>
</tr>
<tr>
<td>4.3 Research methodology</td>
<td>72</td>
</tr>
<tr>
<td>4.3.1 Research site</td>
<td>73</td>
</tr>
<tr>
<td>4.3.2 Research process</td>
<td>74</td>
</tr>
<tr>
<td>4.4 Results and Analysis</td>
<td>75</td>
</tr>
<tr>
<td>4.4.1 Pre-transition</td>
<td>75</td>
</tr>
<tr>
<td>4.4.2 During transition</td>
<td>77</td>
</tr>
<tr>
<td>4.4.2.1 VGT test suite maintenance for improvement</td>
<td>79</td>
</tr>
<tr>
<td>4.4.2.2 VGT test suite maintenance required due to SUT change</td>
<td>80</td>
</tr>
<tr>
<td>4.4.3 Post-transition</td>
<td>82</td>
</tr>
<tr>
<td>4.5 Discussion</td>
<td>85</td>
</tr>
<tr>
<td>4.5.1 Threats to validity</td>
<td>88</td>
</tr>
<tr>
<td>4.6 Conclusion</td>
<td>88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5 Paper D</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>92</td>
</tr>
<tr>
<td>5.2 Background and Related work</td>
<td>94</td>
</tr>
<tr>
<td>5.3 Industrial case study</td>
<td>96</td>
</tr>
<tr>
<td>5.3.1 The industrial projects</td>
<td>97</td>
</tr>
<tr>
<td>5.3.2 Detailed data collection in Case 1</td>
<td>99</td>
</tr>
<tr>
<td>5.3.3 Detailed data collection in Case 2</td>
<td>101</td>
</tr>
<tr>
<td>5.3.4 The VGT suite</td>
<td>102</td>
</tr>
<tr>
<td>5.4 Results and Analysis</td>
<td>103</td>
</tr>
<tr>
<td>5.4.1 Test system related CPLs</td>
<td>104</td>
</tr>
<tr>
<td>5.4.1.1 Test system version</td>
<td>104</td>
</tr>
<tr>
<td>5.4.1.2 Test system (General)</td>
<td>108</td>
</tr>
<tr>
<td>5.4.1.3 Test system (Defects)</td>
<td>110</td>
</tr>
</tbody>
</table>
CONTENTS

5.4.1.4 Test company specific CPLs . . . . . . . . . . 112
5.4.1.5 Test system (Environment) . . . . . . . . . . 112
5.4.2 Test tool related CPLs . . . . . . . . . . . . . . . . 113
  5.4.2.1 Test tool (Sikuli) related CPLs . . . . . . . . 113
  5.4.2.2 Test application . . . . . . . . . . . . . . . . 117
5.4.3 Support software related CPLs . . . . . . . . . . . . 119
5.4.4 CPL Summary . . . . . . . . . . . . . . . . . . . . . . 120
5.4.5 Potential CPL solutions . . . . . . . . . . . . . . . . 121
5.4.6 Defect finding ability, development cost and return on
  investment (ROI) . . . . . . . . . . . . . . . . . . . . . . 126
5.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
  5.5.1 Challenges, Problems, Limitations and Solutions . . 132
  5.5.2 Defects and performance . . . . . . . . . . . . . . . . 134
  5.5.3 Threats to validity . . . . . . . . . . . . . . . . . . . 135
5.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136

6 Paper E . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
  6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
  6.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . 141
  6.3 JAutomate . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
    6.3.1 Tool comparison . . . . . . . . . . . . . . . . . . . . . 143
    6.3.2 Business . . . . . . . . . . . . . . . . . . . . . . . . . 146
    6.3.3 Architecture . . . . . . . . . . . . . . . . . . . . . . . 146
      6.3.3.1 The tool . . . . . . . . . . . . . . . . . . . . 147
      6.3.3.2 The system under test (SUT) . . . . . . . . . . 149
    6.3.4 Process . . . . . . . . . . . . . . . . . . . . . . . . . . 150
    6.3.5 Organisation . . . . . . . . . . . . . . . . . . . . . . 150
  6.4 The industrial need . . . . . . . . . . . . . . . . . . . . . . . 151
  6.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152
  6.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
Chapter 1

Introduction

1.1 Introduction

Software development companies are continuously faced with challenges, such as ever growing demands on software quality and shorter time-to-market. These challenges affect all aspects of software development, from requirements engineering over design and implementation to testing. Testing, which is generally performed using manual practices, aim to verify or validate that the system under test (SUT) conforms to its requirements. These tests are performed on different levels of system abstraction, from low-level component testing, to high-level scenario-based interaction testing through the SUT’s graphical user interface (GUI). However, these practices are costly, time-consuming and tedious, which also makes them error-prone [1–6].

For example, in scenario-based tests that take a long time to conduct, testers sometimes take shortcuts, or do not note faulty interactions. This can thus affect test outcomes and lead to false negative or false positive test results. To make matters worse, requirements and software is prone to change, which requires re-testing to ensure that the changed SUT still conforms to the requirements, i.e. regression testing [7, 8]. Regression testing is as such very important for quality assurance, but each regression test also raises the overall SUT development cost.

In order to mitigate or at least alleviate these problems, e.g. decrease costs and raise quality, automated testing has been proposed as a key solution [3,6,9]. Automated testing allows software developers to raise test frequency, providing quicker feedback to the software developers. It also has the potential to free up resources since the often time consuming system tests can at least partly be run without human interaction. There are many available test automation techniques, e.g. unit testing [10], model based testing [11] and record and replay [12, 13], each with different strengths and weaknesses. In addition, companies create their own tools to fill specific testing needs [2].

However, all of these techniques provide only limited support for automating higher level testing such as system and acceptance testing. For instance, attempts to use unit testing for system testing have resulted in costly, complex, and unmaintainable applications that has spurred an ongoing debate if low-level test techniques are at all applicable for high-level testing [14,15]. Sim-
ilarly, custom-developed test applications also suffer from problems regarding complexity and maintenance [9]. Even record and replay techniques, designed for testing at the GUI level has limitations, e.g. require access to the SUT source code and are sensitive to GUI layout or code change, which result in high maintenance costs [15, 16]. Consequently, there is still need for further research to find an industrially applicable technique that supports cost-effective, flexible, system and acceptance test automation.

In this thesis we present an empirical, and industrial, evaluation of a novel test technique called Visual GUI Testing (VGT). VGT combines scripting with image recognition-based SUT interaction, i.e. interaction with the top SUT GUI bitmap and input layer displayed to the user on the computer monitor [17, 18]. The core of VGT is the image recognition, which, at least in theory, allow more flexibility and makes VGT impervious to many of the limitations of previous GUI-based test techniques. For instance, VGT can be more robust to GUI layout changes, since the image recognition can identify bitmap components regardless of their position on the monitor and even after (smaller) variations in the exact graphical representation. The technique is also black-box and do not need access to or any information about the inners of the SUT. This can make VGT more robust to SUT code changes but also independent of SUT development language or even platform.

However, the core of the technique, i.e. the image recognition-based interaction, is not novel, innovated in the early 90s by Potter and his tool Triggers [19]. However, since image recognition is computationally heavy, it was not until recently that hardware and advances in algorithm technology made the technique practically viable, i.e. mature enough for industrial application [20]. Yet, still VGT only has a limited body of knowledge, e.g. regarding tools such as Sikuli [21], and is only used within a small number of companies. Thus, the main question this thesis aims to answer is if VGT is, actually, mature enough to be applicable for system and acceptance testing in industry. A question that we address through synthesis of evidence collected in three empirical, industrial case studies performed in collaboration with three companies, and using three different VGT tools. In addition, whilst collecting evidence to answer the main question, we also evaluated the challenges, problems and limitations with the technique that may/will affect the transition to, or usage of, VGT in an industrial context. Thus, providing both industrial practitioners and academics with insights into aspects to consider, or further explore, in the future work with VGT.

The continuation of this thesis is structured as follows. This chapter will introduce the thesis, starting with a background to the purpose and need for the research (Section 1.2) followed by a further description of the research focus in Section 1.3. Section 3.3 then describes the research methods that were applied, followed by an overview of the individual studies in Section 1.5. The study overview is followed by a discussion, 1.6 of the thesis results, including a synthesis of results, Section 1.6.1, followed by a discussion regarding the the thesis limitations, Section 1.6.2 and future research, Section 1.6.3. Finally this introductory chapter is concluded in Section 1.7. The rest of the thesis, Chapters 2 to 6, are the individual papers that document the studies.

1Note that there is a difference between GUI testing and system testing through the GUI; we concern ourselves solely with the latter.
1.2 Background and related work

The background to the work presented in this thesis comes from four groups of problems that originate both in industry and academia. The first problem relates to the manual test practices that are currently used in industry for system and acceptance tests. These tests are generally performed using test case scenarios, executed against the system under tests (SUT) [22–24], e.g. through its graphical user interface (GUI). Manual test cases can be of varying length, of varying complexity and ambiguousness, and are executed, under time pressure once every month to once in one year. Because manual tests have these properties, these tests are typically costly, tedious and error-prone [1–6].

The cost comes from the complexity to create tests of high quality and the execution of said tests. Ambiguity in documentation as well as in noticing and understanding system output adds cost by requiring testers to put in more time and effort to fully understand what and how to test. Tediousness, in turn, comes from two parts. First, the length of individual test cases, which, even though complex as a whole, usually contain mundane tasks that are neither fun or stimulating to perform. Second, a manual test suite is generally of a substantial size, especially for complex systems. It is not uncommon that test suites can take several days or even weeks to execute in full. Furthermore, since the test suites are used for regression testing, some testers may be required to execute the same tests cases several times during a development iteration, thus adding to the tediousness of the work. In addition, since system and acceptance tests are usually performed at the end of a development projects, there is often time pressure imposed by impending deadlines. The time pressure, coupled with the tediousness, complexity and ambiguity make the manual tests error-prone, errors caused by testers taking shortcuts in the test cases, not correcting faulty interactions with the SUT in test scenarios, etc. Furthermore, since the tests are performed so late in the development cycle, defects have very little time to get fixed, i.e. raising a need for higher test frequency.

The second overall problem, for the background of this thesis, relates to low-level automated testing, e.g. unit testing. Unit testing is a powerful technique for testing components of a system. Since unit tests are often small and have only localized effects, unit testing can be done often, providing more direct feedback to developers. However, for high-level testing, low-level test techniques are considered, by many, to be unsuitable, even though this is an ongoing debate [2, 15]. Figure 1.1 provides a tree-based visualization that illustrates one of the roots of this problem. In the figure, one input on the top GUI layer stimulates many components on lower levels of the system. For smaller systems, this stimuli and intermediate signals may be traceable, but for larger systems it is often unclear exactly which and how the components are stimulated, especially in systems that generate instances of components at runtime, or concurrent systems. Thus, in order for a test technique that interacts with a system on a lower level of abstraction, e.g. a unit test, to perform a system requirement conformance test in the exact same way as a, for instance, GUI-based manual test case, the unit test would have to emulate the run-time sequence of the high-level test by sending inputs and outputs to the affected components, not only in the right order, but also with a specific timing. Furthermore, because of the lower abstraction level of these tests,
complex functionality, like GUI interaction, they require many lines of code. In contrast, high-level techniques generally use methods of interaction that are tailored to that level of abstraction, which not only makes scripting easier, but maintenance as well, i.e. less lines of code need to be maintained.

Consequently, these properties of low-level tests provides further support that there is a need for research into high-level test techniques for system and acceptance test automation. Furthermore, a higher and more appropriate abstraction level is required to make the tests more maintainable.

The third problem, concerns currently available high-level automated test techniques. There are several high-level test techniques available to the software industry, the most common being coordinate- and component/widget-based record and replay, which will be discussed in more detail in Section 1.2.3. These techniques provide the user with support to automate high-level tests, performed through GUI interaction, thus supporting raised test frequency of system and acceptance tests, release resources such as human testers, etc. Hence, they fulfill many of the requirements of a high-level automation technique. However, there are also several limitations, such as sensitivity to GUI layout and code change, platform dependence, etc., which associates the techniques with considerable maintenance costs and limit their general applicability [3, 9, 15, 16, 25–27]. Consequently, even though these previous techniques partially fill the gap for a high-level test automation technique, there are still areas of improvement that warrant further research into alternative, better, system and acceptance test automation techniques.

The fourth and final problem relates to the second problem, i.e. that there is need for more high-level test automation techniques in industry. Visual GUI
Testing (VGT) [17, 18] could be the solution to meet this need. However, the body of knowledge regarding the technique is very limited, especially in terms of empirical work that supports the technique’s industrial applicability. Hence, there is need to evaluate the technique to find evidence that can support or reject the technique’s applicability in industry, and thereby its ability to meet the current industrial needs.

In summary, the background to the work performed in this thesis originates in the fact that manual test techniques are costly, tedious and error-prone, low-level automated test techniques’ applicability for high-level test automation is questioned, currently available high-level automaton techniques include many limitations, and finally, the work on VGT is close to non-existent. Below we give further background to the main areas on which this thesis builds.

1.2.1 System and Acceptance testing

Software testing is performed in many phases during the development of a system, with different purposes in each phase. In the early stage of a development project, testing is generally restricted to component level tests performed through unit testing [10]. Once a larger set of components have been developed, these components are integrated into larger components, subsystems or the complete system. During integration it is common practice to perform integration testing to verify that the components work together without failures. Once the components have been integrated, and a large amount of functionality has been developed, a system test can be performed. The purpose of a system test is to ensure that all the system’s components work together, but more importantly that the system complies with its requirements. Requirements are artifacts that define some need that a customer, or user of the software, has that the system should fulfill [28, 29]. However, system testing only aims to verify requirements conformity, which we separate from validation of the requirements. Validation is usually performed by the actual customer, or a customer proxy, and in addition to ensuring requirements conformity also aims to ensure that the system behaves as the customer intended, and as expected in the system’s intended domain [28,29]. Thus, validation is performed during acceptance testing of the system.

System testing and acceptance testing have many commonalities, i.e. they are both high-level, usually scenario driven, e.g. with use cases [22–24], aim to test the system in its entirety, or at least larger and coherent portions of it, and are usually based on the system’s requirements. However, with the difference that acceptance tests are customer driven whilst system tests are usually driven by testers or developers at the development company. Quite naturally, because developers are seldom experts in the customer’s domain, the developers often lack the detailed domain knowledge that is needed to perform validation and thus to create and select inputs for acceptance tests.

There are different types of system tests, with different purposes performed on different levels of system abstraction [29]. However, generally, they are done on higher levels of system abstraction since the purpose is to verify the system as a whole. Thus, much of the testing, in systems that have some kind of GUI, is performed through the GUI using test scenarios that aim to stimulate a part or parts of the system. These scenarios can be documented in different ways,
i.e. in tables, as use cases, etc. [22–24], where each scenario is built from a sequence of test steps followed by one or more assertions about the expected system output or behavior. Thus, each test step defines some input for which there is, usually, a deterministic, and expected, output. Figure 1.1, on the left, shows how a single click on the SUT’s GUI can propagate through the system, i.e. stimulating several components and methods and thereby some percentage of the source code and the system’s functionality. This percentage of stimuli is thus related to the notion of test coverage [30]. There are different types of coverage, branch coverage, statement coverage, component coverage, etc., but in a system or acceptance test the focus is generally on function coverage, i.e. does all the functions of the system work according to expectations [31]. To achieve coverage of large software systems a very large number of test scenarios might be needed, i.e. a large test suite.

Large test suites are associated with several problems, primarily high cost, since it takes a long time to develop/create, perform, and maintain all of the test cases. Furthermore, if the test scenarios are long, which can be required in a complex system, the test cases also become tedious to perform, which leads to mistakes and shortcuts to be taken during test execution.

In addition, most test scenarios are based on fixed input, i.e. specific values such as numerics or letters, which only ensures that the system works for said input. However, in real-world use, any input values may be acceptable, which is infeasible to test manually due to time-constraints. Instead, the scenario is generally performed with a set of predefined values that test one, or a few, common cases for which the system should work. Automated techniques exist to solve or mitigate this problem through automated test case generation, e.g. test cases generated from models.

Hence, we can conclude that manual testing of a complex system is a difficult task that suffers from several problems. First, manual testing is tedious, costly [1, 2] and in addition, due to the need for full function coverage, often also complex. Especially since these tests, preferably, should also cover a practically infinite input space, which is generally impossible. Thus, when taking these aspect into consideration, the need for test automation becomes quite clear. However, at the same time, studies have shown that manual testing will always be required, and that it is important that the testing is performed on many levels of system abstraction [15, 32]. Furthermore, manual testing, e.g. exploratory manual testing, due to the cognitive element, supplied by the human tester, has many benefits over many automated testing techniques, such as the ability to find faults incidentally in parts of the system not being tested, making use of experience, domain knowledge and SUT knowledge that cannot be automated, complex faults can be identified using simple measures, etc [33].

1.2.2 Automated Testing

There are many types of test automation techniques. Some commonly discussed, and/or used, techniques in industry or academia are unit testing [10, 14, 30, 34–36], model based testing [11, 37–40] and record and replay (R&R) [2, 12, 13, 15, 16, 20, 27, 41, 42]. Since the R&R techniques are the most similar to the subject of this thesis, i.e. Visual GUI Testing, they will be discussed
Automated testing is generally performed in two steps. First some input is given to the SUT, after which one, or many, assertion(s) are performed to verify the SUT’s output with the help of an oracle, where the oracle either has access to, or can evaluate, an expected output based on the input [43–45]. Oracles can be more or less advanced, the least advanced being based on simple value comparisons, whilst the most advanced rely on close to cognitive reasoning, i.e. artificial intelligence. However, the purpose of automated testing is to increase test frequency and lower cost, since automated tests can be performed much quicker than manual tests and at almost no cost. Thus, providing the developers with quicker feedback on the quality of the SUT, and faults that require fixing. Furthermore, the low cost of automated test execution also allows for data-driven testing, i.e. execution of the same test scenarios several times but with different input. Hence, providing a solution to the problem posed for manual testing regarding testing of the entire input-space. However, all automated testing is associated with an upfront costs, such as tool costs, education of users costs, development costs of the automated test cases and their oracle, etc [46,47].

The automated test, and oracle, development costs are generally higher than just performing the test case once, or even several times, manually. Therefore, test automation should only be performed if the automated test is expected to be executed several times against the SUT, e.g. for regression testing. [7,8,47] The purpose of regression testing is to verify that the system still conforms to its requirements after a change has been done to the system, i.e. that changes to one component of the system has not affected the behavior of components in other parts of the system. Regression testing is therefore an important practice, which has also been adopted into many processes, e.g. agile processes such as eXtreme Programming (XP) [48]. In XP, regression testing is performed to facilitate continuous integration, most often using automated unit testing [10].

Unit testing is performed on a lower level of system abstraction, i.e. component or method level, by writing statements of input and output that test the component or method out of context. Hence, a input value is given to the component after which an assertion is performed against an expected output. If the output from the component matches the expected outcome, the assertion has passed. However, as mentioned, due to the fine granularity of unit testing, it is difficult to write unit tests that test larger parts of the system at once, i.e. system tests [2,15]. It is especially difficult to create assertions to verify graphical output, i.e. GUI tests, since the output is generally the generation of a bitmap that may be stochastic. Hence, unit tests are limited in their applicability to test high-level tests, but a powerful tool for component level testing.

Another common automation technique, as stated, is Model Based Testing (MBT) [37]. The core of MBT is to use a model of the tested system, which can be constructed in many ways, e.g. using UML, state-diagrams or event flow graphs [40,49,50]. These models are then used to automatically, or semi-automatically develop test cases, or the models are used as oracles. The use of MBT to create test cases helps to mitigate two of the problems discussed in Section 1.2.1, i.e. that it is hard to cover the often exceedingly large input
space and acquire full test coverage of a software system. By using models, several thousand or even millions of test cases can quickly be developed and executed against the SUT, either using more basic brute force solutions or more advanced solutions. One advanced solution is based on search-based techniques, where the optimal set of test cases can be generated using a set of search criteria, e.g. coverage criteria, etc.

In order for any MBT to be useful and successful, the models, just like test cases, continuously have to be maintained such that they align with the system and/or the system requirements. However, many MBT techniques are complex, e.g. based on advanced mathematical principles, which require expert knowledge to be used to great effect which also raises cost for development and maintenance [50, 51]. Furthermore, most MBT techniques perform automated tests by calling methods and components directly from within the source code. These MBT techniques are therefore limited in terms of applicability to SUTs with specific properties, i.e. specific programming languages, architectures, etc. However, there are also higher level techniques, which are more general, including MBT techniques for GUI-based test case generation, automated GUI-model creation, maintenance and test execution [50]. Thus, providing cost-effective support for higher-level tests.

1.2.3 Automated Graphical User Interface Testing

Many automated test techniques that are commonly used in industry, e.g. unit testing, approach testing from lower levels of abstraction, which, as we have discussed, perceivably limit their use for high-level testing. However, there are also techniques that are based on high-level test automation, i.e. against the GUI layer of a system, with the most common referred to as Capture and Replay or Record and Replay (R&R) [2, 12, 13, 15, 16, 20, 27, 41, 42]. There are two main types of this technique, with different benefits and drawbacks. The benefits are that these techniques can perform close to human emulation, test non-functional properties such as performance, etc. However, they also suffer from limitations such as sensitivity to GUI layout and code change, platform dependencies, etc. Furthermore, these different techniques have been classified into two different generations based on, when in time, they were introduced. The first generation relies on Coordinate-based interaction with the SUT, whilst the second generation, being more advanced, relies on software component/widget-based SUT interaction. In the following sections these two types or R&R are presented in more detail together with their independent benefits and drawbacks.

Note that there is a difference between GUI testing and system testing through the GUI. In the continued discussion we focus on the latter.

1.2.3.1 Coordinate-based Record and Replay

Coordinate-based Record and Replay (R&R) belongs to the first generation of GUI-based test techniques [2]. As clear from its name the technique consists of two separate steps. First, a user’s interaction with the SUT is stored, using a tool, in a recording, e.g. a database or a script. Second, the R&R tool replays the recording to give the same system use and stimuli through the GUI, but now automatically. In this approach, the user’s interaction with the SUT is
recorded based on the coordinates that the user clicks on the screen. Keyboard interactions with the SUT are also recorded. In addition, the tools capture the system state during recording which can then be used to create assertions for the replay phase.

However, this coordinate-based approach has several disadvantages, the largest being that the scripts become very sensitive to GUI layout change. Hence, since this approach relies on the absolute x and y coordinates on the SUT’s GUI, moving any widget or button on the GUI will cause the script to fail \[2, 16\]. Thus, this technique is maintenance heavy, requiring either maintenance of the coordinates in the scripts or even re-recording of the scripts, even for small GUI changes. Hence, because the recordings are so sensitive, the technique has limited use for regression testing \[25\].

Furthermore, since an automated script should have a life-expectancy, i.e. number of executions before requiring maintenance, which surpasses the cost of the automation in order to be cost effective, as mentioned by Finsterwalder and Malte and Marick and Brian, this technique is generally not cost-effective \[3, 47\].

However, at the same time as the technique is sensitive to GUI layout change, it is more robust to changes of the software design and code. These changes can include changing the properties of GUI object, i.e. software components that determine the look and feel the GUI bitmap object shown to the end user. Thus providing support for the technique’s use for regression testing given that the GUI layout remains unchanged. In addition, due to the exact coordinate approach, the technique is very fast. However, since the scripts can’t run quicker than the GUI can interact, the technique requires delays to be added in the scripts to synchronize the script execution with the SUT. This is a common problem for all GUI-based techniques, which will be mentioned further in this thesis.

Consequently, coordinate-based R&R is the first generation of GUI-based test techniques, but which suffers from limitations in terms of robustness, which has ultimately resulted in research and development of other approaches to perform R&R. However, due to the flexibility provided by the coordinate-based interaction, i.e. interaction anywhere on the screen, this approach is often incorporated into more modern GUI-based testing tools, which will be discussed in the following sections.

1.2.3.2 GUI component-based Record and Replay

The second generation of GUI-based test techniques is also referred to as a record and replay technique, i.e. Component- or Widget-based Record and Replay. This technique builds on the same concepts as the first generation technique but instead of, or in addition to, using coordinates, it extracts the properties of the GUI components, e.g. their name, id or type, to uniquely identify and interact with them on the SUT’s GUI. Thus, making this technique more robust to GUI layout change than the first generation R&R technique since GUI components can be identified regardless of their position. Furthermore, component-based R&R is performed with the same steps as its predecessor, i.e. a recording is performed during manual interaction with the SUT, which can then be played back automatically, from some type of record-
In order to add assertions to the tests, these tools use what is referred to as verification points [13]. A verification point is a snapshot of the state of the SUT after a set of interactions have been performed against it. These snapshots can be taken by capturing the properties of all GUI components at a certain time \( t \), or, which is supported in some tools, by taking a screenshot of the GUI. The saved state or images can later be used to verify that the system is still giving the proper responses. For the image-based verification points this requires image comparison algorithms.

Together, these approaches make component-based R&R more robust [2], but with the main drawback that it often requires the tool to have access to the source code of the developed application to perform any interaction. There are however tools, e.g. TestComplete [52], which circumvent this need by hooking into the Windows operating system, but even these tools suffer from limitations that limit their use for certain applications.

One of these limitations, common to most component-based R&R tools, is that they use the properties of standardized GUI components to identify components on the GUI, e.g. properties such as name, identification number, color information, etc [13]. However, many GUIs are built from custom components, components that are drawn in runtime directly onto a canvas, or components that are even generated during runtime. Since these components are not in the format expected by the tools, or don’t exist prior to script execution, they cannot be interacted with [3, 26]. Some tools therefore provide the user with the ability to construct custom interaction procedures, which the user can then call during recording. These custom procedures can, as an example, define custom properties, or architectural templates, of the custom components that the tool’s should identify, or require, during recording and playback. However, properties of a GUI component may, and often do, change during software development, causing scripts to fail since the sought properties of a component no longer align with the script, thus requiring maintenance [27]. This also requires any custom interaction procedures to be maintained. Furthermore, even though several R&R tools can interact with applications written with different programming languages, many are restricted to specific programming languages, e.g. Java. Thus, for applications that are cross-language, or distributed cross-platform, or distributed on several physical computers, these tool’s generally do not work.

However, in contrast to the previous coordinate-based technique, component-based R&R is impervious to GUI layout change. In many cases these tools can even detect, and interact, with widgets that are hidden on the screen [26]. In some instances this capability is positive, but it also has an inherent drawback, i.e. that the test case execution differs from how a human user would perform the interaction. This differentiation is also inherent in most tools that deploy this technique because the SUT interactions are performed by calling widgets’ click methods rather than using the operating system’s actual click functionality. Thus, a test case executed automatically with this technique
1.2. BACKGROUND AND RELATED WORK

can pass, whilst the same, equivalent, manual test case might fail, e.g. if a button is hidden, R&R tools can still invoke the click, whilst the human user wouldn’t. Furthermore, interaction directly with widget methods also makes the technique susceptible to code or API change, which in some cases has caused entire test suites to become unusable when a system has been migrated from one toolkit to another, e.g. from Java 1.4.2 to 1.5 [53]. Other studies have shown that even small changes to the SUT can require changes to the test scripts, i.e. 30 to 70 percent of the scripts require maintenance only from small code or GUI layout changes [9, 15, 16]. In addition, this approach makes the technique intrusive, which becomes especially apparent in tools that require special hooks to be added to the SUT’s source code. To install the hooks one is often required to change the internal structure of the tested system. Furthermore, similar to the coordinate-based technique, this technique also requires the script to wait for the SUT to reach a stable state after an interaction has been performed, i.e. delays need to be added into the scripts.

To summarize, component-based R&R is a more robust technique in terms of script execution, but it has several limitations, including that most tools require access to the source code and is thereby intrusive, can be sensitive to code or API change, etc. Furthermore, common to both R&R approaches, due to the record functionality, the techniques requires the system to be in a more mature state before any scripts can even be recorded, i.e. both the system and its GUI needs to have been developed [13]. Thus, this technique also suffers from limitations that hinder its usability, flexibility and cost benefit.

1.2.4 Visual GUI Testing

Visual GUI Testing (VGT) is a novel technique that combines the strengths, such as flexibility, from image recognition with the beneficial properties of script-based high-level testing [17,18,20]. However, the core of VGT, i.e. using image recognition in GUI interactions, is not novel but where proposed in the early 90s by Potter in his tool Triggers [19]. Other early work from the late 90s include Zettlemoyer and Amant with their tool VisMap [54]. Thus, the technique has existed for many years, but as we have previously mentioned, it has still not been industrially applied in any larger scale. The main reason for this is because image recognition is resource intensive, and it is not until recently that hardware has become powerful enough to make this technology viable [20].

VGT differs from previous GUI-based test automation techniques in several regard. However, as stated, the main difference is its image recognition capabilities, which allow the user to create scripts that can interact with any bitmap component, e.g. image, button or text field, shown on the computer monitor. In addition, all VGT tools interact with the SUT in the same way a human user would and can do so regardless of implementation language, architecture or even platform of the SUT. As long as the SUT has some kind of visual presentation interface a VGT tool can, in principle, interact with it. This gives VGT unprecedented capabilities such as testing a single system written in several programming languages or using several rendering or GUI component frameworks, systems distributed across hardware platforms, e.g. desktop and mobile devices, as well as operating systems. Existing VGT tools
<table>
<thead>
<tr>
<th>Property</th>
<th>Coordinate-based Record and Replay</th>
<th>Component-based Record and Replay</th>
<th>Visual GUI Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent of system under test (SUT) platform</td>
<td>- -</td>
<td>- -</td>
<td>++</td>
</tr>
<tr>
<td>Independent of SUT programming language</td>
<td>++</td>
<td>+/- -</td>
<td>++</td>
</tr>
<tr>
<td>Non-intrusive</td>
<td>- -</td>
<td>+/- -</td>
<td>++</td>
</tr>
<tr>
<td>Emulate exact human user behavior</td>
<td>++</td>
<td>- -</td>
<td>++</td>
</tr>
<tr>
<td>Open-source tool alternatives</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Supports manual test case automation</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Supports testing of custom GUI components</td>
<td>++</td>
<td>+/- -</td>
<td>++</td>
</tr>
<tr>
<td>Supports bitmap-based assertions</td>
<td>- -</td>
<td>+/- -</td>
<td>++</td>
</tr>
<tr>
<td>Supports testing of distributed systems</td>
<td>++</td>
<td>+/- -</td>
<td>++</td>
</tr>
<tr>
<td>Robust to GUI layout change</td>
<td>- -</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Robust to system code change</td>
<td>++</td>
<td>- -</td>
<td>++</td>
</tr>
<tr>
<td>Support script recording (as opposed to manual scripting)</td>
<td>++</td>
<td>++</td>
<td>+/- -</td>
</tr>
<tr>
<td>Robust to bitmap GUI component change</td>
<td>++</td>
<td>++</td>
<td>- -</td>
</tr>
<tr>
<td>Script execution time independent on system performance</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
</tr>
<tr>
<td>Replacement to other manual/automatic test practices</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
</tr>
</tbody>
</table>

Table 1.1: The positive and negative properties of different GUI-based test techniques. All properties have been formulated such that a “++” indicates a positive property and a “- -” indicates a negative property. “+/- -” indicates that some of the technique’s tools supports the property, but most don’t.
typically support the most common computing platforms, i.e. Windows, Mac OS, Linux, etc.

The technique is tool-driven, with tools such as Sikuli [20,21], Eggplant [55] and JAutomate [56]. The former two tools require the user to write test scripts manually and then allow references (Eggplant) to or the actual images (Sikuli) to be matched and interacted with to be inserted into scripts. JAutomate is different and combines a R&R approach with image recognition. Thus it supports recording of scripts which are then more flexibly executed through the use of image recognition [56]. The benefit of the record and replay functionality is that it allows the user to quicker, i.e. at lower cost, develop scripts that can then be replayed using the strengths of the image recognition.

Most VGT tools also use fuzzy image recognition algorithms, which help make them robust, i.e. minor discrepancies between the target bitmap and the source bitmap are tolerated. Furthermore, because of the image recognition, VGT is non-intrusive, meaning that it does not require any modification, or even access to the SUT, in order to be applicable. In addition, VGT is impervious to both GUI layout and code or API change since interacts with the system on a highest level of abstraction.

However, as a main drawback the technique is instead sensitive to changes to the bitmap components, such as change of color, scale, etc. Additionally, like previous GUI-based test techniques the performance of VGT is governed by the performance of the SUT, i.e. a VGT script can not run quicker than the SUT can respond to the script interaction. Thus, VGT scripts are required to be synchronized with the SUT, which can be done through visual verification that the SUT has reached a stable state before the next interaction is performed.

In summary, VGT overcomes some of the core limitations of previous techniques, e.g. VGT is impervious to GUI layout and code change, it is non-intrusive, and highly flexible. However, even though there are several benefits to the technique it is still not widely used in industry. One reason can be the technique’s immaturity, i.e. the technique is not yet widely known. Second, the lack of tool-support, and rather high costs of commercial, mature, tools that do exist, which discourages companies from testing them. Third, companies used to previous GUI-based techniques may have invested heavily in their current test suites and are therefore reluctant to change. Furthermore, the question regarding the techniques industrial applicability still remains. The answer to this question is the primary goal of this thesis, and the continued discussion therefore aims at providing support for such an answer.

1.3 Research Focus

The goal of this thesis is to evaluate if Visual GUI Testing (VGT) is applicable for automation of system and acceptance tests in industry. However, applicability is broad term, and the contributions of this thesis are therefore delimited to showing VGT’s technical applicability on industrial grade test cases, short-term project applicability and short term, initial, maintenance issues. Consequently the long-term applicability, and feasibility in terms of cost, are outside the scope of the thesis. The reason for this delimitation is because the maintenance costs of VGT’s industrial use have not been thoroughly eval-
uated and are therefore a subject of future work. Also, by its very nature, maintenance costs cannot be easily studied in a shorter time span, as has been available to us in this project.

Hence, the aim of this thesis is to find evidence to answer the two research questions,

- **RQ1**: Is Visual GUI Testing applicable in industry for system and acceptance test automation?

- **RQ2**: What are the challenges, problems and limitations of transitioning industrial system tests into, and using, Visual GUI Testing in industry?

Due to the thesis strong focus on empirical and industrial work, the thesis also contains a secondary focus, based around how to perform and explore a research phenomena broadly in its industrial context. This secondary focus was investigated first and therefore provided knowledge regarding different research methods, how these can be applied in empirical research, and thus provided support for how to answer the thesis two main research questions. The secondary focus was evaluated through a meta level study based on the support research question,

- **SRQ**: How can an exploratory case study be conducted to capture the research phenomena from multiple perspectives?

The acquired results to answer this question resulted in a research framework, presented in Chapter A. However, its interest for this thesis primarily lies in its presentation of research methods, which will be presented further in Section 3.3.

In summary, the main research focus of this thesis regards the industrial applicability of VGT, supported through a meta-level research framework for conducting multi-perspective evaluation of research phenomena.

### 1.4 Methodology

The following section will start by presenting the fundamentals of a research methodology and how the methodology affects the quality, validity and reliability of the research results. However, the core of the section will present and discuss the research methods that were used in the work presented in this thesis. For methods that can be performed in different ways, each has been named an only briefly summarized to keep focus on the methods that were used. A summary of the used methods is shown in Figure 1.2 together with what type of research they were used for, i.e. explorative, descriptive or validatory research. In addition, the figure also shows the research type of the individual studies.

#### 1.4.1 Research Methodology

A research methodology is a systematic process, of steps, practices and methods that when executed should yield a sought research result. These process steps are phases of the research study, or objectives, in which a research method or methods are used to observe, evaluate, explore, etc., a research phenomena.
The foundation of a research methodology is as such the research focus, i.e. what we wish to study in the context of the phenomena, defined through hypotheses and research questions. A hypothesis is a statement with a null hypothesis and an alternative hypothesis that can either be supported or rejected by a research study, whilst a research question is a question that the study should answer. This answer generally constitutes one of the study’s research contributions.

Research methods, in turn, are the tools, such as observation, interviews, surveys, etc., which researchers use to gather evidence to support or reject the research hypotheses and answer the research questions. Different research methods have different strengths and weaknesses, dependent on the research focus and context. Hence, selecting the right research methods is key for the research to gain high quality, validity and reliability.

1.4.2 Research Validity and Reliability

The validity and reliability of a research result refers to if we can trust the research results, and claims, such as stated contributions and conclusions, regarding the results, i.e. might our results be wrong? [57] There are many types of research validity, concerning different aspects of trust [58]. Some of these types are external, internal and construct validity and reliability. External validity concerns the generalizability of the results, contributions and conclusions, i.e. do they apply in other contexts, companies, etc. Internal validity concerns the logical, causal, relationships of the results, i.e. does the collected evidence support or contradict each other. Construct validity, concerns the research methodology and if it actually measures what was intended to be measured, i.e. does the measured results actually help answer the research questions. Finally, reliability concerns the replicability of the research, i.e. in order for research to have high reliability it should be possible for other researchers to replicate it and/or the results.

1.4.3 Experiments

Experiments are controlled studies that aim to evaluate or observe a subject matter, the dependent variable, of, or within, a phenomena and its context. The context is important because it includes the independent variables, which we can measure or control to observe or manipulate the dependent variable. A simple example of a dependent variable could be water temperature, which we can regulate through the using the knobs on the water faucet, i.e. the independent variable. Furthermore, the context also includes confounding variables, i.e variables that we cannot control, identify, account, or care for, but which can influence the dependent variable and therefore affect the experimental results. A confounding variable, from our previous water faucet example, would be that we measure the temperature using an old quick-silver thermometer which affects the precision of the results.

An experiment where all variables are controlled is called a controlled experiment, achieved through random sampling of the studied phenomena. Random sampling assures that confounding variables, with statistical probability, cancel each other out, leaving only the dependent and independent variables
that the researcher can control/wish to measure [59]. Hence, controlled experiments are artificial in nature, which affects their external validity, but at the same time they have inherent high internal validity because of the controlled causal relations between independent and dependent variables. Experiments where not all variables can be controlled, e.g. in industrial experiments, are called quasi-experiments, which due to the lack of control have lower internal validity but higher external validity than true experiments [60].

In Paper B, a true experiment was used in a pre-study to evaluate the applicability of VGT on different types of GUIs. The experiment was performed by executing scripts written in two different VGT tools on a random selection of applications with different GUI properties, i.e. animated and non-animated GUIs. The research method was chosen because it provided both broader and deeper understanding of VGT’s image recognition capabilities, at low cost, since the experiments could focus only on the image recognition. Furthermore, these image recognition capabilities were unknown prior to the experiments, but had to be understood to perform the case study in Paper B. Quasi-experiments could have been used as an alternative method, i.e. on systems provided by our industrial collaborators, but it had been more costly and would not have significantly improved the results.
1.4.4 Interviews

Interviews are a versatile research method that can be used in descriptive, explanatory and experimental research designs. There are three main types of interviews, which have different naming conventions in different literature, but which we will refer to as, structured interviews, semi-structured interviews and open interviews [61].

Structured interviews is the most strict form of interviews, performed with an interview guide which the interviewer may not stray from, even to ask follow up questions. These interviews are generally performed with several people, i.e. the interview guide is reused, with the benefit that the answers from the interviews can be compared and analyzed using statistical methods. The drawback with this technique is that it is reliant on a qualitative interview guide in order to produce qualitative results, which requires a lot of work to formulate good questions, evaluation of said questions, deep understanding of the phenomena under study, etc.

Semi-structured interviews are a less strict form of interviews, also performed with an interview guide, but with this method, the interviewer is allowed to ask follow up questions to the interviewee. The benefit with this method is that it allows the interviewer to go deeper on subject matters of interest, using follow up questions, but with the drawback that the answers from different interviewees become more diverse and therefore harder, or impossible, to compare.

The least strict form of interviews are called open interviews, performed more as discussion between the interviewer and interviewee. This method is beneficial in contexts where very little is known about the phenomena under study, e.g. in exploratory studies, but with the drawback that it requires experience in behalf of the interviewer to produce good results.

Semi-structured and open interviews were used in the studies presented in Papers A and D. The study presented in Paper A was explorative, therefore, semi-structured interviews were used to get a better understanding of phenomena under study, i.e. the state-of-practice of an industrial software developer. Semi-structured interviews were used in favor of open interviews because basic understanding of the state-of-practice in general industry was acquired before the study started, but the details of the studied company were unknown. Therefore, the basic understanding could be used to formulate interview guides, whilst specific subjects of interest could be probed deeper using follow-up questions.

In the study presented in Paper D however, no information was previously available regarding the studied phenomena, i.e. neither the studied companies state-of-practice or use of GUI-based testing, and therefore open interviews were first used to get basic understanding. Once a more concrete understanding had been acquired, semi-structured interviews were used to gain deeper insights. Specifically, the semi-structured interviews were used to validate previously acquired information from the company and to probe subjects of interest further, e.g. the company’s experiences with VGT, once again using follow-up questions. Furthermore, these interviews were deep interviews, meaning that they were performed with an excess of 65 interview questions, including four fundamental questions aimed at eliciting the industrial practi-
tioners’ perception of VGT.

Surveys, which will be covered in Section 1.4.5, could have served as an alternative to the semi-structured interviews in both studies, but had limited the ability to probe deeper on subjects matters of interest, even if open ended survey questions, i.e. free text, questions had been used.

1.4.5 Surveys

Surveys are studies performed with sample groups of individuals to draw general conclusions regarding the groups or the population, from where the sample was taken. Surveys are performed differently, depending on research domain, but for Software Engineering interview and questionnaire surveys are the most common approaches.

Interview surveys are performed as structured interviews, see Section 1.4.4, following interview guides. The benefit of this approach is that all survey participants will answer the survey, but the drawback is that it is very expensive and time consuming.

Questionnaire surveys are more cost-efficient, i.e. can be sent out to a large sample group at almost no cost. Drawbacks with this approach include that it can be problematic to get high enough response-rate, qualitative, unambiguous questions can be difficult to formulate, etc. However, to aid researchers, there are guidelines for how to write questionnaires, e.g. using Likert scales [62]. Questionnaire’s reliability can also be evaluated using Cronbach alpha [63]. The alpha value defines how coherent the questions are and thus provides a value of reliability of the questions. Preferably this analysis is done after a pre-study, where the questionnaire is tested, before the actual survey is sent out to the sample group. It should however be noted that if the alpha value is low, indicating that one, or several, questions should be changed, a new pre-study needs to be performed to verify that the changes have had an effect.

Questionnaire surveys were used in the studies presented in Papers A and E. In Paper A a questionnaire was used, consisting of both Likert scale forced choice, and open ended questions, to elicit state-of-practice knowledge from the industrial practitioners at the studied company. The choice of this method, with this approach, was because the industrial practitioners were too many to interview and the understanding of the state-of-practice had previously been evaluated using other methods. Hence, the survey was used for validation of said understanding.

In Paper E, the purpose of the questionnaire was to explore state-of-practice and state-of-art in industry regarding test-related problems and the use of VGT. The questionnaire consisted of a combination of Likert scale questions and questions defined using the 100 dollar method [64]. This survey was conducted during a seminar with 100 industrial practitioners, which would not have been possible with any other technique due to cost and time constraints.

Hence, for Paper E, no alternative existed to the questionnaire survey, whilst in Paper A an alternative approach could have been to use interviews, even though it had been a lot more costly.
1.4.6 Workshops

Workshops, similar to focus groups [65], are performed with groups of participants, e.g. industrial practitioners, where the subject of research is discussed, analyzed or evaluated, e.g. to perform static validation [66]. Workshops can be performed in different ways, for instance with seminars followed by discussion, brainstorming sessions, practitioners working with a tool or method under study, etc. The main benefit of the approach is that a lot, and deep, information can be acquired in a short time-span. Drawbacks with the technique, because it is performed in a group environment, is that participants with authority can overrule suggestion from other participants, participants may act differently than in a normal working setting, etc. Furthermore, workshops require a lot of planning and experience from the workshop leader in order to be successful.

This method was used in Paper D, where two workshops were held, on site, at the studied company during which information about the studied company’s processes, the system they were developing, manual testing, etc., were elicited from the industrial practitioners. The first workshop was used as a project start-up with the studied company, during which discussions and brainstorming were used to acquire information, but also to define future research objectives. In the second workshop, focus was on information validation, performed through group-based open discussions with the industrial practitioners, paired with, as stated in Section 1.4.4, semi-structured interviews. The workshop method was chosen for this study since a lot of information had to be acquired during a very small time frame. Furthermore, this method, in contrast to for instance interviews, gave the industrial practitioners the opportunity to contribute to the research design, i.e. in the first workshop.

Alternative methods that could have been used would have been pure semi-structured interviews in combination with document analysis. However, that had limited the two-way communication provided by the workshop approach.

1.4.7 Other

In Paper A, two additional methods, other than the previously stated, were used. However, since they were only used to a small degree, and were not used later during the thesis, we only briefly summarize them below.

Observations were used in the study performed for Paper A since it was explorative and focused on assessing, and describing, the software development practices used at a company. This method is performed following an observation guide that contains items that are expected to be observed during the session. Thus, if an item in the guide is observed, said item is crossed off the list. Additional observations, not anticipated in the guide, can be added during observation. There are different kinds of observations, denoted with different naming conventions such as structured observation, unstructured observation, open observation, etc [67]. The defining character of a structured observation is that the researcher cannot interact with the individual or process being observed. However, with open observation the observer is allowed to ask questions to clarify what the observed individual is doing, but this also affects the results since the questions interrupt the observed individuals workflow.
In Paper A, open observation was used to gain deeper knowledge into how the test activities at the studied company were performed. The reason why open observation was used in favor of structured observation was because the researcher had limited knowledge about the process and the tested system. Thus, the researcher had to ask clarifying questions to ensure that all new items, added to the observation guide, were valid. Structured observation could have been used instead, coupled with a follow-up interview, but due to time constraints this was not possible in this situation.

The second method used in Paper A, was the so called watercooler discussion method. Watercooler discussion is a form of open interview that is performed in the industrial context, provided that the researcher has access to the company under study, by asking one or several open ended questions to the industrial practitioners during a break. The term, Watercooler discussion, originates from the United States, where watercoolers are common, at which, individuals meet and talk over a glass of water, i.e. short interactions during a break. The benefit with the method is that it allows the researcher to acquire the interviewees’ real views on a subject, since it is not performed in a strict interview environment. The more relaxed “watercooler environment”, allows the interviewee to be more open, but with the drawback that he/she might present very biased views on the subject matter. Thus, this method can be dangerous for the research validity, and all information acquired using this method must therefore be triangulated using other research methods or sources of information. Furthermore, the method requires the researcher’s discretion, since there might be ethical concerns how the elicited information is used, i.e. the researcher should anonymize any and all information.

The technique was, as stated, used in the study for Paper A, and to great effect helped identify aspects of the studied company’s process that could later be verified through triangulation with deep interviews. Alternative methods that could have been used would have been questionnaire surveys to acquire broad information, coupled with semi-structured interviews to acquire deeper information on specific subject matters. However, this had been more costly, and it is uncertain if the interviewees views would have been captured in the same way.

1.5 Chapter/Study Summaries

This section will summarize the studies that were conducted for each of the papers included in this thesis, including the research goals, questions, context, results and contributions. The focus will be on an individual paper level of abstraction, synthesis of the results will be presented in Section 1.6.1.

1.5.1 Chapter 2: Measuring a research phenomena from multiple perspectives

Chapter 2 presents Paper A, titled “Structuring Software Engineering Case Studies to Cover Multiple Perspectives”, which presents a case study with the research goal of identifying the state-of-practice at a safety-critical software development company. However, rather than to present the study results, the
paper presents the meta-level research framework for conducting exploratory case studies that was developed and used during the study.

The framework, called the BAPO/PCF framework, is a multi-perspective framework that provides practices for conducting, and managing, an exploratory case study at a company. In addition, the framework presents a core artifact called the BAPO/PCF matrix, which consists of two axes. On the first axis, the y-axis, the BAPO (Business, Architecture, Process and Organization) dimensions are used to provide a broader view of the studied company, or phenomena that is being studied at the company. By studying multiple dimensions there is less risk that a single focus or perspective takes precedence. This helps in creating a more holistic view of the company and its processes and gives a richer basis for finding explanations.

The second axis of the BAPO/PCF, the x-axis, represents time as Past, Current and Future (PCF). Thus, the purpose of the matrix is to help the researcher identify information both in a cross-sectional and a longitudinal fashion. Hence, in addition to identifying the current state-of-practice, the framework focuses on change, i.e. what was the state before and how did the company end up in the current state, and how may this transition influence the future. Consequently, the framework is suited for conducting initial studies where little is known about the company context, but where the context is important for future research.

In addition, the paper presents how the researcher can formulate cause-effect chains across the matrix, i.e. between cells and states of the studied phenomena. These cause-effect chains constitute chains of evidence that link the cells, or states, of the matrix together, i.e. how information found in one cell influence information gathered in another. Thus, helping the researcher to understand how the phenomena has gone from one state to another, which also raises the internal validity of the acquired information by logically, and chronologically, linking the information together [58]. In addition, these chains can be visualized within the matrix to reveal patterns and help present the gathered information and its analysis. Furthermore, the BAPO/PCF matrix helps the researcher visualize how triangulation has been performed, e.g. between different information as well as different research methods by simply stating which methods were used to acquire information in what cell. Thus, raising the construct validity of the research.

Hence, the main contributions of Paper A are:

[A] A framework for conducting exploratory case studies called the BAPO/PCF framework.

[B] A detailed record of the advantages and disadvantages of using the framework, as well as its impact on research validity.

Because of the research focus on this thesis, these contributions are considered secondary, i.e. not part of the main contributions of this work. However, the framework, and included practices, have influenced the work performed for the main research track and therefore warrants Paper A’s inclusion in the thesis.
1.5.2 Chapter 3: Visual GUI Testing in practice, initial support

Chapter 3 presents Paper B, titled “Automated System Testing using Visual GUI Testing Tools: A Comparative Study in Industry”, which presents an experimental study performed at the Swedish safety-critical software development company Saab AB, subdivision SDS in Gothenburg. The research goal was to do an initial evaluation of Visual GUI Testing (VGT) in an industrial context to identify its applicability for system test automation for regression testing.

To evaluate the technique, and not just a tool, two VGT tools were used in the study and compared against one another based on both their static properties, e.g. number of image recognition sweeps per second, number of image recognition algorithms and script language, and dynamic properties, e.g. ability to automate industrial manual system test cases. The two compared tools were a commercial tool, called CommercialTool due to confidentiality reasons, and an open source tool called Sikuli. The study was performed in two phases. In the first phase, an analysis was first performed to identify the tools’ static properties, mentioned above. In addition, a series of academic experiments were conducted to identify the tool’s ability to interact, and test, different types of Graphical User Interfaces (GUI), i.e. animated and non-animated GUIs.

In the second phase, the tools were used to automate five carefully chosen, manual test cases for a safety critical air traffic management system at the company. These five test cases constituted 10 percent of the entire manual test suite for the system. During the automation and execution of the test cases, quantitative metrics were collected on development time, execution time and lines of code in each script, as well as qualitative metrics on advantages and disadvantages of VGT, and the tools, compared to manual testing. The collected information was then analyzed, including statistical analysis of the collected quantitative metrics. Results of the analysis showed that Sikuli was able to interact better with both non-animated and animated GUIs than CommercialTool. In addition, Sikuli also had a more powerful scripting language, which was also estimated to have higher learnability than the language used in CommercialTool. Furthermore, the analysis showed that both tools were equally applicable in the industrial context, able to automate all the selected test cases, with no statistically significant difference between the collected metrics. Additionally, in terms of the tool’s static properties, e.g. image recognition sweeps per second, the tools were not deemed to have any significant differences. The main difference between the tools was the cost, which also resulted in Saab choosing Sikuli rather than CommercialTool. Whilst Sikuli was free of charge, CommercialTool had a cost of roughly 10.000 Euros per license per year, not including initial costs for development of a working test suite. Maintenance cost of the test suite is also not represented.

Hence, the main contributions of Paper B are,

[A] Initial support that Visual GUI Testing can be applied by knowledgeable researchers to successfully automate industrial grade test cases with both a commercial VGT tool and a open source VGT tool.
A detailed record of the advantages and disadvantages of the two tools, shown from both an academic experimental context and an industrial context.

Hence, this study was pivotal for the main research track of this thesis and provided initial support for VGT as a technique, which prior to this work was a gap in knowledge within academia. Furthermore, this research introduced VGT to industry and has since its publication lead to application of the technique in several companies. Thus, this research has had impact on both academia and industry.

1.5.3 Chapter 4: Visual GUI Testing in an industrial project

Chapter 4 presents Paper C, titled “Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study”, which presents a case study, which is part experimental and part descriptive, conducted at the company Saab AB, subdivision SDS in Järfalla. The research goal of the study was to show that Visual GUI Testing (VGT) is applicable in industry when performed by industrial practitioners, under real-world time and cost constraints. Previous empirical work, i.e. Paper B, was also conducted in industry, with an industrial system, but driven by academics with deep knowledge in using VGT. In contrast, the study in Paper C aimed at capturing the practitioners’ perception of VGT, whilst collecting both qualitative and quantitative information regarding costs, problems and solutions experienced/identified during the VGT implementation at the company.

The study was driven by two industrial practitioners, testers, at the company on a safety- and mission-critical military control system. Both practitioners had deep knowledge about the system, the domain, and the extensive manual test suite for the system, which consisted of roughly 4000 use cases structured in 40 acceptance test descriptions (ATDs). Out of the 40 ATDs, three were chosen for automation based on their complexity, i.e. two were considered more complex than the remaining ATDs and the last was considered equal in complexity as the remaining ones.

Information acquisition was performed during two full day workshops, on site at the company, where the first workshop aimed at identifying the current state-of-practice at the company, SUT properties, etc. The second workshop was used for validation, where deep, semi-structured, interviews were conducted to triangulate previously collected data. Between the two workshops the testers also supplied the research team with their progress, metrics, problems, etc., over email and through telephone communication on a regular basis. A structured data acquisition process was put in place, but due to time constraints, experienced by the industrial practitioners, this process could not be followed. Hence, most of the collected information was qualitative, even though some quantitative metrics were also acquired for the implementation costs, bugs found, Return on Investment (ROI), and maintenance costs of the developed VGT suite. In particular, the metrics regarding maintenance costs of the VGT suite were of interest, since detailed information regarding VGT’s maintenance costs are still missing to draw a conclusion regarding the long-term industrial applicability of VGT. Maintenance cost information could be
collected because a huge maintenance effort was performed on the SUT three calendar months into the four month project, a maintenance effort that caused 90 percent of the VGT suite to break. The cost to get the entire VGT suite working again, which included changing between 5-30 percent of all images in all test scripts, was evaluated to 25.8 percent of the VGT suite’s development cost.

Results from the study also showed that the VGT tool that was used, Sikuli, included many problems, limitations and also bugs. Even so, the industrial practitioners stated that they had not found any test that they could not automate, and that the benefits of the automation clearly outweighed the experienced issues. For instance, the industrial practitioners reported that, due to the automation, the system test frequency could be increased from one manual execution every six months to several automatic, equivalent, executions every week. In addition, due to the higher test frequency, the VGT suite was able to identify three, previously unknown faults in the SUT that the testers themselves stated would never had been uncovered if it hadn’t been for VGT.

Hence, the main contributions of Paper C are,

[A] Support that Visual GUI Testing (VGT) is applicable in industry when performed by industrial practitioners.

[B] Initial support on the positive Return on Investment of using VGT as well as manageable costs for implementation and maintenance of VGT scripts in a real-world industrial context.

[C] Detailed descriptions of VGT challenges, limitations and solutions that were identified during the automation of a safety- and mission-critical military control system.

Hence, the results from this study compliment the results from Paper B by providing support for the real-world application of VGT when implemented, and applied, by industrial practitioners. However, even though the study found evidence to suggest that VGT suite maintenance costs are manageable, this is still a subject of future work.

1.5.4 Chapter 5: The challenges, problems and limitations of Visual GUI Testing

Chapter 5 presents a Paper D, titled “Visual GUI Testing in Practice: Challenges, Problems and Limitations”, which presents the synthesis of information, acquired from two studies performed in industry, where manual test cases were transitioned into automated test cases using VGT. In addition, the study had a particular focus on the challenges, problems and limitations (CPLs) related to VGT. Synthesis was performed with additional, previously unused, information from the study presented in Paper C, called Case 2, together with information acquired from a second study performed at Saab AB in Gothenburg, called Case 1. Thus, the main focus in Case 1 was to identify CPLs related to the transition to, or usage, of VGT when performed with the open-source tool Sikuli. Information about the CPLs was acquired by expanding on our previous work, presented in Paper B, through automation of the remaining test cases from the test suite used in our previous work. Additionally,
a secondary focus of the study, i.e. for both Case 1 and Case 2, was to find possible, generic, solutions to the identified CPLs.

In order to capture the CPLs in a systematic way, a detailed data acquisition process was put in place in Case 1, where the researcher performing the automation documented all CPLs that were encountered, together with detailed quantitative metrics on development time, script lines of code, etc. During the study, 58 CPLs were identified, which were then categorized, based on the CPL's origin, into 29 mutually exclusive groups of CPLs. These 29 groups were divided further into eight more generic groups, which were either related to the specific version of the test system, the test tool (Sikuli), third party software, the test system in general, defects in the test system, company specific, the test system environment or the test scripts. Analysis of the distribution of the CPLs showed that 34 of them were related to the test system itself, whilst only 14 were identified to the tool. The remaining 10 CPLs related to support software such as the virtual network connection and third party recording software. Hence, most of the CPLs that were uncovered were context dependent.

Most of the information regarding the CPLs were taken from Case 1, but was corroborated by information from Case 2. Most of the 29 groups of CPLs were described, followed by a summary of the most important CPLs and their effects on Robustness, Usability, Portability, Reusability, Maintainability and Modifiability of the test scripts. Furthermore, the paper presents four generic solutions that can solve, or mitigate, roughly 50 percent of the identified CPLs. Only four solutions were identified since focus was on finding solutions to the non-context dependent CPLs. These solutions include adding redundant script failure mitigation code in the scripts, perform thorough script documentation, selective use of local or remote script execution and finally adoption of practices for systematic SUT synchronization of the VGT scripts.

Finally, the paper presents quantitative metrics on development cost, execution time, and Return on Investment (ROI) from both of the evaluated cases. However, since the two studies were performed in different contexts, automating manual test cases with different architecture, etc., most of the presented information can not be compared between the two cases. What can be determined however, as also stated by the industrial practitioners, is that the development cost was not considered unreasonable in either case, and that the positive ROI can be acquired quickly once the automation has been completed, perceivably within on development interaction. Furthermore, the results provided support for the technique’s fault-finding ability, with identification of 9 defects in the two systems, of which 5 where previously unknown to the studied companies.

Hence, the explicit contributions of Paper D are,

[A] A detailed record of 29 mutually exclusive groups of a total of 58 identified challenges, problems and limitations (CPL) that affect the implementation or application of VGT when applied in industry.

[B] A detailed record of four solutions to roughly half of the identified CPLs that, perceivably, are generic to any industrial context.

[C] Metrics regarding development time, execution time, bug finding ability, and return on investment from two industrial projects, as well as a high-
level comparison of said data that indicate the general feasibility of VGT when applied in an industrial context.

Hence, Paper D builds on previous work and contributes to the previous collected information regarding VGT’s application and implementation costs in industry. Furthermore, the identified challenges, problems, and limitations (CPL) outline future work regarding VGT, since these CPLs need to be mitigated either through technical innovation or implementation of new, more formal script development practices. In addition, some of the identified CPLs are perceived to be generic for all automated testing. Thus, providing a valuable contribution to the, currently limited, general body of knowledge regarding automated testing related CPLs [68].

1.5.5 Chapter 6: JAutomate, a tool for Visual GUI Testing

Chapter 6 presents paper E, titled “JAutomate: a Tool for System- and Acceptance-test Automation”, which presents the tool JAutomate, a Visual GUI Testing tool supplied by a Swedish consultancy company called Inceptive AB. The tool, in contrast to most other VGT tools, supports record and replay of VGT-based scripts. Hence, it enables the user to record scripts during manual execution of the manual system and/or acceptance tests.

In the paper, JAutomate’s properties are compared to two other VGT tools, i.e. Sikuli and CommercialTool, based on information from Paper B. The comparison shows that JAutomate has several characteristics in common with the other tools, e.g. several image recognition algorithms, support for developing and managing test-suites, etc., whilst the main difference, as stated, is the tool’s record and replay functionality. Other than the previous mentioned record and replay support, JAutomate supports both manual test step and semi-automated test step execution, which can be used to drive manual test cases within the automatic script suite, i.e. parts or entire manual test cases can be incorporated into the automated tests. In addition, similar to CommercialTool, JAutomate supports creation and maintenance of test suites from within its IDE. Furthermore, the tool has been designed to be incorporated in other testing frameworks, e.g. FitNesse, thereby making JAutomate a very test oriented tool, whilst the other two focus on more general GUI automation.

Paper E also presents an analysis of the tool’s perceived impact on the BAPO aspects of a company, BAPO standing for Business, Architecture, Process and Organisation, described further in Paper A. This analysis showed that JAutomate will primarily impact the Business, Architecture and Process aspects, whilst Organization will not be affected due to the simplicity of the tool, i.e. it has been designed with simplicity and usability in mind in order not to require any specific roles or expert support. Finally the paper presents the results of an industrial survey aimed at identifying the industrial need of the tool and the test related problems that currently appear in industry. Analysis of the survey results showed that there are several test related problems that JAutomate can solve and that 71 percent of the participants in the survey would investigate JAutomate further, i.e. had an interest in the tool.

Hence, the explicit contributions of Paper E are,
1.6 Discussion

This section will present a synthesis of the results and contributions from the individual papers and discuss how they connect to each other. An overview of these connections is shown in Figure 1.3 which also relates the papers to the research questions and contributions. Furthermore, we will discuss limitations with our studies and results as well as future work regarding Visual GUI Testing.

1.6.1 Contributions

In this thesis we have provided evidence that Visual GUI Testing (VGT) is an industrially applicable technique for high-level test automation. We have also shown that automated VGT scripts can raise the frequency of high-level tests without additional execution cost, whilst retaining, or even improving, the fault finding ability of the tests. In addition, we have found evidence that suggests that the development, and maintenance costs, of VGT can be manageable, but conclude that this subject requires more work. Furthermore, we have acquired evidence that there are many challenges, problems and limitations (CPLs) related to VGT that warrant future work and require both technical- and process-oriented improvements and solutions.

The core of Visual GUI Testing is the use of image recognition that in combination with scripting allows the technique to interact with, and test, any graphical user interface (GUI) based software system. This technique was innovated already in the 90s by Potter in his tool, Triggers [19]. However, it is not until recently that advances in hardware performance, and better algorithms, has made the technique viable for industrial use and lead to the creation of several VGT tools, e.g. Sikuli [20], EggPlant [55] and JAutomate [56]. The academic body of knowledge contains very little work on this technique, leaving a gap in knowledge regarding if VGT is industrially applicable.

Exploration of this gap in knowledge is warranted because current manual test practices, used for high-level tests, have properties that make them costly, tedious and error-prone [1–6]. Properties that outline a second gap in knowledge and a need for new automated test techniques, like VGT. Further support to this need is given from literature, where different automation techniques, e.g. unit testing and Record and Replay, have been used to try and cover the

[A] The results of a qualitative comparison between JAutomate, Sikuli and CommercialTool that shows that the tools have different properties that make them more or less suitable in different contexts.

[B] Results of a survey that shows that there is an industrial need, and want, for JAutomate, and Visual GUI Testing, in Swedish software industry.

Consequently, this paper compliments the results from Paper B by expanding the comparison from that paper with a third VGT tool, i.e. JAutomate [56]. Furthermore, the paper provides support from the trenches, through a survey with 52 industrial practitioners, that there is a need and want for VGT in industry.
CHAPTER 1. INTRODUCTION

RESEARCH QUESTIONS

RQ1: Is Visual GUI Testing applicable in industry for system- and acceptance-test automation?
RQ2: What are the challenges, problems and limitations of transition ing industrial system tests into, and using, Visual GUI Testing in industry?
SRQ: How can an exploratory case study be conducted to capture the research phenomena from multiple-perspectives?

CONTRIBUTIONS

C1: Initial support for the feasible implementation and application of several Visual GUI Testing (VGT) tools, and thereby the technique, in industry.
C2: A detailed record of the challenges, problems and limitations of Visual GUI Testing when performed with the open source tool, Sikuli.

Figure 1.3: Visualization of how the independent studies, performed during the thesis, link together.

second gap, each technique with specific strengths and potential for different types of testing. However, due to limitations in the techniques, none of them have been able to fully bridge the gap, e.g. creating assertions for high-level tests such as GUI tests with unit tests is difficult, coordinate-based record and replay tools are sensitive to GUI layout change and component/widget-based record and replay is sensitive to code or API changes [2, 3, 9, 15, 16, 25–27].

The first gap, regarding Visual GUI Testing, was bridged by Papers B and C that could show that VGT is an industrially applicable technique. Initial evidence that support the technique’s applicability was acquired in the study presented in Paper B, where two different VGT tools were used, and compared, by knowledged academics to transition manual, safety critical, industrial test cases, into VGT tests. The comparison showed that there were no significant differences between the tools therefore provided support for the technique’s industrial applicability rather than just the applicability of any one VGT tool. Furthermore, the studies presented in Papers B and C showed that VGT tests have equal, or even higher, fault finding ability than the manual test cases they were created from, adding further positive support for the technique. In addition, the study presented in Paper C, which was performed by indus-
trial practitioners, showed that VGT is also applicable in an industrial project development environment, i.e., under real-world conditions such as resource constraints and with evolving/changing software systems.

However, as reported in Paper D, there are also many challenges, problems and limitations (CPLs) related to VGT. These CPLs hinder the application of the technique but the industrial practitioners that took part in the study still stated that the technique is both beneficial and cost-effective in practice. Thus, providing yet further support for VGT, even though the CPLs must be considered in the holistic picture. A holistic picture that Paper E also added to by comparing previous results to a third VGT tool, i.e., JAutomate, showing its industrial applicability as well.

Furthermore, Paper E showed, through a survey with 52 industrial practitioners, that the second gap, i.e., the need for high-level test automation techniques, still warrants further research, e.g., with VGT. This thesis provides the fundamental building blocks to bridge this gap, but it does not cover it all the way. The limitations of previous GUI-based techniques, e.g., Record and Replay, mainly concern their high maintenance costs, maintenance costs which, as we have reported, are still unknown, and therefore a subject of future work for VGT. Initial evidence that the maintenance costs for VGT are manageable was presented in Paper C, but more research is required before a conclusion can be drawn regarding VGT’s long-term applicability for system and acceptance test automation in industry.

Hence, the explicit contributions of this thesis are,

[A] Initial support for the feasible implementation and application of several Visual GUI Testing (VGT) tools, and thereby the technique, in industry.

[B] A detailed record of the challenges, problems and limitation of Visual GUI Testing when performed with the open source tool, Sikuli.

Furthermore, Paper A presented a multi-perspective framework for conducting exploratory case studies. This work has influenced the overall work into VGT, thereby warranting inclusion in this thesis. Hence, providing the thesis third contribution,


1.6.2 Limitations

The following section will discuss the limitations of the results presented in this thesis.

There are two main threats to the validity of the research presented in this thesis. The first threat stems from the sample of companies that have taken part in the performed studies, i.e., Saab AB in Gothenburg and Järfälla. Both of these companies develop safety-critical, as well as mission-critical, software that has been under development for many years. Furthermore, the systems have, and are, primarily being developed using iterative or plan-driven processes in longer projects. These aspects have both positive and negative effects on the external validity of the research. Thus, even though the studies help provide good external validity for the industrial applicability of VGT for safety-critical software system development companies, little or nothing can
be said about smaller agile development companies. Hence, the contributions of this work may be generalizable also to agile projects, but given that agile projects have a general characteristic of being more volatile, e.g. rapidly changing GUIs and code, the maintenance of the VGT scripts may outweigh the benefits. Furthermore, only two companies were used for the core studies of this thesis, which also hampers the external validity. Even so, due to the similar findings, these results are still regarded as significant.

The second threat regards the fact that most of the studies used the open source tool Sikuli. Thus, a majority of the critical findings, i.e. regarding development cost and initial metrics on maintenance costs, were captured only for said tool. Thereby, the external validity of these findings are limited in regards of what the actual cost may be for any generic VGT tool. However, as presented in Paper B, which included a comparison between Sikuli and CommercialTool, we saw that there were no statistically significant differences between the script development costs between the two tools. Thus, raising the internal validity and construct validity regarding the conclusion that VGT is feasible and applicable in industry. Furthermore, Paper E, presented a comparison between Sikuli, CommercialTool and a third tool, JAutomate. The comparison showed that JAutomate had several differences to the other two tools, but no significant difference in terms of industrial applicability. Hence, supporting the conclusion of this thesis regarding VGT’s general industrial applicability.

However, in terms of challenges, problems and limitations (CPLs), i.e. the second research question of this thesis, the results are once again primarily limited to Sikuli. Hence, lowering the external validity of the results regarding the CPLs. It is however perceived that many of the presented CPLs from this work, Paper D, should be generalizable because of their generic nature, not just to VGT but to automated testing in general.

In addition, even though the methodology used for each study has been presented in detail, the fact that the studies were performed in the real world has negative effects on the reliability of the research. Hence, it is doubtful that the studies can be completely replicated, even though the hypothesis is that they should be replicable, given similar circumstances, and context, in the companies where the replicating studies would be performed. This hypothesis is further supported by the previous discussions that indicate that the work presented in this thesis has high construct validity and external validity for safety-critical software companies. In addition, the internal validity is also perceived to be high, even though it may be lowered because of the properties of said context, i.e. safety-critical software development companies are generally large and complex, which makes it harder to account for all confounding factors.

The discussion above has only covered the main research track, i.e. regarding VGT, from the thesis. For the supporting track, i.e. regarding the BAPO/PCF framework, the validity is overall low. However, the construct-validity can be considered moderate, because BAPO is a re-occurring set of perspectives in academia. Furthermore, using a longitudinal perspective is equally common in academic work. For external validity little can be said however, because the framework has, to the author’s best knowledge, only been applied in one case so far. Even though the guidelines of the framework
Table 1.2: Summary of validities and reliability for each of the thesis research questions. The scale used in the table ranges from Low to Moderate to High. **RQX** - research question X, **SRQ** - support research question

has influenced the overall work presented in this thesis, it has not been explicitly used in its presented form. Future, planned, work does however aim to use the framework further, thereby providing further support for its usefulness. However, currently, the framework’s empirical usefulness can only be speculated upon, even though it was used to great effect in the study where it was conceived.

A summary of the internal, external and construct validity, as well as reliability, has been presented in Table 1.2 for each of the thesis research questions. Note that the validity levels are based on the synthesis of all results and contributions from all papers in the main research track, i.e. regarding VGT’s industrial applicability, for RQ1 and RQ2, but only the contributions from Paper A for SRQ.

1.6.3 Future Research

Based on the synthesis of the results from the conducted studies, presented in Section 1.6.1, it can be concluded that Visual GUI Testing (VGT), has industrial applicability, also when performed by industrial practitioners. Due to these results, there are also many venues of future work regarding VGT that explore different aspects of the technique’s beneficial properties, e.g. it’s flexibility. However, as previously stated, the work presented in this thesis has only presented that VGT can be applied in industry. Hence, the thesis has not explored the long-term feasibility of the technique, nor other potential venues of VGT-related research. However, as presented in Figure 1.4, we have been able to outline five main areas of future work regarding VGT, i.e. five tracks, which are the,

- Fundamental track
- Alignment track
- Process track
- Psychology track
- Technical track

The fundamental track will be discussed in more detail in Section 1.6.3.1 due to its importance, whilst the remainder of this section will concern the remaining four tracks.
CHAPTER 1. INTRODUCTION

Detailed VGT costs:
1. Maintenance
2. Implementation
3. Development
4. Management

VGT scripts from requirements. Automatic script generation.

Technical innovation to mitigate and solve problems with VGT

Figure 1.4: Possible directions of future work regarding Visual GUI Testing, where the fundamental track is the most important.

Alignment track: VGT is a script-based technology, and even though there are VGT tools that allow the user to quickly, and perceivably cost-effectively, record scripts during manual interaction with the system under test (SUT), the development costs cannot be ignored. Furthermore, VGT scripts aim to test the SUT’s adherence to its requirements, which shows an inherent connection, and need for alignment, between the SUT’s requirements and the VGT scripts. Thus, one area of future research, based on the practices of Behavioral driven development (BDD), would be to create a framework for writing structured requirements that could be used to automatically generate test scripts. Hence, minimizing the need for costly, and potentially error-prone, manual script development, or even recording.

Process track: As shown by this thesis, there are many challenges, problems and limitations (CPLs) connected to the transition into, and application, of VGT. In addition, the VGT transition is associated with an initial cost, which potentially could be mitigated through a structured, more effective script implementation process. Hence, the creation of such a process is another subject of future work, which includes not only the creation, but also evaluation of such a process, for industrial use. Factors of specific importance, to consider in such work include, but are not limited to, that the transition process has to be light-weight, safe guarded against implementation that is not warranted within the studied company, minimize the amount of required resources, and
more. Furthermore, in order to ensure long-term viability of such a framework it should be designed such that it can be iteratively improved and generalised to different contexts, i.e. both small, medium and large companies.

**Psychology track:** VGT is perceived to mitigate some of the tediousness experienced during manual regression testing since it can, to some degree, replace the human tester, allowing said tester to focus on more explorative testing. However, in order for VGT to achieve this, the VGT suites need to be up to date and complement the exploratory testing by ensuring qualitative, automatic, regression testing. Ensuring that the scripts are of high quality, and up to date, is however a non-trivial task, which might add stress, lack of motivation in using the technique, etc. Consequently, there are several soft, psychological, aspects to consider regarding VGT, e.g. do developers/testers actually enjoy working with the technique? Hence, evaluating these factors, and their effects on the quality of the testing and the software as a whole, is another subject of future research.

**Technical track:** As previously stated in this section, there are several CPL’s related to VGT, some of which relate to the technical limitations of the VGT tools, e.g. limitations of the image recognition algorithms and lack of supporting functionality within the tools. Some of these limitations relate to lacking maturity in the tools, which can be solved or mitigated through new technical solutions, e.g. future development of the tools. Furthermore, as discussed during the alignment track, a perceived future direction for VGT is to generate VGT scripts automatically, which could be solved through technical innovation. Additionally, some limitations have been identified regarding VGT’s ability to automate user interactions with the new generation of touch screen devices, e.g. the users exact, yet, inconcentric patterns, as he/she drags and drops objects on a touch screen. Thus, another technical need refers to better ways of capturing the user’s interaction with the SUT, with particular focus on current, and future, human-machine interfaces.

### 1.6.3.1 The long-term feasibility of Visual GUI Testing

As discussed, there are many directions, and opportunities, for future research regarding VGT. However, before any of the above stated tracks can be evaluated, an answer has to be found regarding the feasibility of long-term applicability of VGT. This subject is still an open question since the maintenance costs of VGT suites are still unknown. Furthermore, since previous GUI-based test techniques, i.e. Record and replay, have been shown to be very costly to maintain, this subject is of particular, and crucial, interest. Future research into VGT therefore has to be devoted to finding exact cost models for the maintenance, implementation, development and management of VGT suites, preferably within different development contexts, with different tools. Thus, this work requires further empirical, industrial research, where VGT suites are both developed and maintained in real industrial projects, during real world time and cost constraints during a longer period of time.

Hence, only once these aspects regarding cost have been thoroughly evaluated can a definitive answer regarding VGT’s long-term industrial applicability be given. Current, initial, results indicate that the maintenance costs could be feasible, i.e. the maintenance of 90 percent of a test suite, written in Sikuli,
could be maintained to full operation within 26 percent of the development cost of said test suite. However, since this result was captured only within one case, in one context, more data is required.

Consequently, there are many potential areas of future research for VGT, but before any of said areas can be explored, more explicit, and thorough, cost models have to be developed for the usage and maintenance of the technique.

1.7 Conclusions

In this thesis we have evaluated and found evidence to support the applicability of Visual GUI Testing (VGT) in industry, but also that there are challenges, problems and limitations that require future work. The topic of this thesis stems from an industrial need for more techniques that support automated system and acceptance testing, since current industrial practices for high-level testing are primarily manual, costly, tedious and error-prone [1–6]. Automation has been proposed as a solution to these problems, but current techniques, e.g. automated unit testing and Record and Replay, all suffer from limitations, e.g. GUI-based testing with unit tests is difficult and record and replay techniques are sensitive to GUI layout or code change. VGT is perceived not to suffer from these limitation but even though the core idea of the technique, i.e. combining image recognition with system testing, have existed since the early 90s, both the technique’s body of knowledge and its use in industrial practice are limited.

In this thesis we try address this gap by answering two research questions. The first question regards the applicability of VGT to automate system and acceptance tests in industry and the second regards what challenges, problems and limitations exist that limit the transition to or use of the technique.

The first research question was answered through empirical studies performed in industry where three available VGT tools were applied, compared and evaluated, by both academics and software practitioners, on industrial projects. Results of these studies showed that VGT is industrially applicable and able to automate manual system test cases with equal, or even better, fault-finding ability than their manual equivalents. Furthermore, the studies provide evidence that the implementation costs, and to some degree also the maintenance costs, of VGT are manageable. However, the acquired evidence regarding these costs are not yet conclusive. Since VGT’s long-term viability in industry depends on it being cost effective we propose this to be the main subject for future work.

Our empirical studies also answered the thesis second research question by identifying challenges, problems and limitations that affect the adoption and usage of VGT in industry. Thus, even though, as stated by the industrial practitioners, some of these challenges, problems and limitations can be overcome using specific practices and work-arounds, many still warrant further work.

To support the empirical work performed during the thesis, a third, supporting research question was also evaluated regarding how to measure a studied phenomena to gain a multi-perspective understanding of it. Evidence acquired to answer this question resulted in a framework that has, and will continue to, influence our future research on VGT.
Hence, this thesis has shown that VGT is industrially applicable but also that further research is required to provide definitive support regarding the technique’s long-term feasibility in industry. Furthermore, the thesis opens up several new areas of future research, regarding both technical issues as well as development practices and processes. In summary, VGT is a promising new automation technique that warrants both more academic and industrial attention in the future.
Chapter 2

Paper A

Structuring Software Engineering Case Studies to Cover Multiple Perspectives

E. Börjesson, R. Feldt

Abstract

Case studies are used in software engineering (SE) research for detailed study of phenomena in their real-world context. There are guidelines listing important factors to consider when designing case studies, but there is a lack of advice on how to structure the collected information and ensure its breadth. Without considering multiple perspectives, such as business and organization, there is a risk that too few perspectives are covered.

The objective of this paper is to develop a framework to give structure and ensure breadth of a SE case study.

For an analysis of the verification and validation practices of a Swedish software company, we developed an analytical framework based on two dimensions. The matrix spanned by the dimensions (perspective and time) helped structure data collection and connect different findings. A six-step process was defined to adapt and execute the framework at the company, and we exemplify its use and describe its perceived advantages and disadvantages.

The framework simplified the analysis and gave a broader understanding of the studied practices but there is a trade-off with the depth of the results, making the framework more suitable for explorative, open-ended studies.
2.1 Introduction

The case study is an observational research method used in many different fields of research due to its flexibility and its ability to investigate a phenomenon in its context [69]. Case studies are also applicable when there is no clear distinction between the phenomena and its context. This is particularly true in empirical software engineering research where there are many factors that impact the phenomenon, such as the type and organization of the company, the development processes used etc. To understand a contemporary phenomenon we also need to understand its history and how the different factors have evolved over time.

Existing advice for empirical research in software engineering focus on experiments and systematic reviews while guidelines for case studies was only recently published by Runeson and Höst [69]. There, a high quality case study is defined as a study that produces valid information of academic or industrial significance, either generic or practitioner oriented [69]. A key criteria for achieving this is that the data is collected in a planned and consistent manner and that conclusions are based on a clear chain of evidence. This can be a challenge in practice since case studies are typically flexible research designs with multiple sources of evidence; it is not clear-cut how to find the right balance between a flexible research design that allows multiple factors and causes to be taken into account while providing enough structure and support for planning and analysis.

Software engineering is different from computer science in that it takes more perspectives than only the technical into account. For example the personality or motivation of the engineers [70,71] can affect the quality of their work and organizational and career considerations can affect important activities such as effort estimation [72,73]. In general, the characteristics not only of the people but of the organization, business and processes used in software development are all important.

In recent empirical research in industry we were faced with designing a case study to describe and understand the verification and validation practices at a Swedish company developing safety-critical software systems. From initial talks at the company we understood that the practices could not be studied in isolation; they were heavily tied to the whole context of the company as well as how they had evolved over time. To structure our data collection and understanding we based our case study on an analytical matrix combining these two main aspects. On one dimension we wanted to cover at least the main perspectives of the BAPO framework with its four aspects [74]: Business, Architecture (technical aspects), Processes and Organization. The other dimension was time detailed in three steps as Past, Current and Future. Together this created a matrix of 12 different sub areas to be considered during the case study. We also defined a general six-step process to design, collect and analyze the findings of the study and adapted it to the company. This paper describes this analytical framework, called the BAPO/PCF framework for software engineering case studies, covering both the matrix and the process to adapt and use it. We exemplify and evaluate the framework based on our application of it at the studied company. In particular this paper addresses the following research questions:
[A] What are the perceived advantages of a multi-perspective approach to conducting a case study?

[B] What are the perceived disadvantages of a multi-perspective approach to conducting a case study?

[C] How does the multi-perspective structure of the design affect research validity?

The definition of a research design is in this paper a methodology used to conduct a research project, such as a case study, an experiment, a longitudinal research project, etc. A research method is defined as a way of eliciting information within a research project; hence a research design can include the use of one or several research methods.

To help the reader differentiate between what was done in the state of practice analysis and what are general case study practices in the paper, we refer to the latter as the ‘case study’ and the former as either ‘company study’ or ‘study at the company’.

The paper is divided as follows, section 2.2 will present the research context in which the research design was developed and executed. Section 3.3 will present how the framework was developed and executed. In Section 4.4 the advantages and disadvantages of the design will be discussed, and finally some conclusions will be presented in Section 6.6.

### 2.2 Research Context and the studied Company

The project, for which the research design was developed, was a state-of-practice analysis of a small company, less than 50 employees, which develop safety critical software applications. The state-of-practice analysis constituted the first part of a larger project, with the goal of improving the company’s verification and validation practices. The analysis was conducted in order to understand the company’s needs, regarding the company’s processes, organization, etc, to narrow the scope of the process improvement effort.

The company conducts software development in bespoke projects to single customers, but the end applications are developed from a set of core products. These products are maintained according to market demand, making the company’s business strategy a mix of bespoke and market driven engineering. Because of the nature of the software the company is developing, the company’s business is governed by different quality standards and frameworks. These standards affect the development, but primarily the company’s verification and validation practices. Standards and frameworks are also imposed on the company by the company’s customers, who have different needs in their own domains. These demands require the company to be flexible in their development, which has resulted in the adoption of iterative as well as an agile software development based on Scrum [75]. The architectural granularity of the company’s systems is on a sub-system level, hence coarse grained, but efforts are made to increase the granularity by refactoring the systems with reusable components based on services, so called service oriented architectures.
(SOA). The decision to change the system architectures was taken internally at the company but driven by external business factors related to a large European project that will change much of the company’s market domain. The migration from sub-systems to SOA has required the company to acquire both new knowledge as well as new practices. Organizational changes have also been made, including changing the responsibilities of different roles and how communication is handled internally at the company. The company study was conducted over approximately 6 months, during which 7 structured interviews, 2 surveys, 1 structured observation, and a considerable amount of hours were spent on document analysis and watercooler discussion.

2.3 The BAPO/PCF Framework

The term research process is in this paper defined as the six-step methodology that was used to develop the case study design used at the company as well as the execution of the design. Figure 3.3 presents the process steps and how they incrementally relate to one another.

Get Knowledge about the Domain. Very little was known about the company when the project started and therefore the first step of the process was to gain a deeper understanding about the company and the company context. This information was acquired through a combination of interviews with people at the company and through literature review of internal documentation as well as documentation about the company’s domain. The context in which the company operates was important to understand the goals and needs of the company, in order to align the improvement effort with these goals, such as process changes or introduction of new tools, etc. The initial interviews also served as a way of finding new sources by using snowballing, sources that could be used later during the company study. Snowballing is conducted by asking questions at the end of interviews and looking at document references to locate further artefacts to study [76]. The majority of the domain documentation that was studied was market quality standards and frameworks. Development and quality standards were important to study because the results of the company study would be used as a base for a larger project that focused on the company’s verification and validation (V&V) practices, hence changes done to the V&V practices would, other than to comply with company needs, also have to comply with such standards. Domain specific documentation is often hard for someone from outside the domain to understand, so therefore all interpretations and question marks related to the analyzed domain documentation had to be verified with people at the company.

The collected data from step 1 showed that a broader view would be required to capture the state-of-practice at the company in its context. To meet this requirement and structure the research a multi-perspective framework was developed based around an analytical matrix with two dimensions. The first dimension was chosen to represent the perspectives of the company that would be analyzed, defined as company Business, Software Architectures, Processes and Organization (BAPO). These perspectives were chosen because they are used within academic and industrial frameworks such as the Family evaluation framework, which is used to evaluate Software Product Lines [74]. The second
2.3. THE BAPO/PCF FRAMEWORK

The BAPO/PCF framework was chosen because it was perceived to be a good fit in this particular study, which might not be the case in the context of another company.

Develop focus Questions/Areas. The second step of the process was to use the information that was gathered in the first step, combined with the BAPO/PCF framework, to narrow down the focus of the company study by developing concrete research questions. The research questions were split up among the 12 sub areas/cells of the analytical matrix, which gave an initial understanding of where within the company to elicit information to answer the research questions. For instance it gave a coarse-grained view of which roles within the company that should be interviewed and what documentation to analyze. Information about existing company roles and documentation was provided by the first step of the process. The research questions were split among the 12 matrix cells based on the researchers own opinion on where they fit best, with the company context taken into consideration. For instance questions regarding the company’s agile development processes were mostly constricted to the process row of the matrix, only sorted in time, whilst questions about the organization were split up between business, processes and organization in this case. The reason for the spread of organizational questions came from the need to understand how the organization had been affected by the other company aspects, for instance how the introduction of agile processes had affected the organization, or what affect business changes had resulted in, how roles had changed, etc. Some questions could not be confined to just one cell, such questions were mostly of higher level, but they helped to find logical connections between cells within the matrix. The options in these cases were to add the question in several of the cells, or to redefine them to make each of the questions unique and only fit in a specific cell.

Choice of Detailed Research Methods. The third step of the process was to
choose what research methods that would be used in the company study. Primarily qualitative methods were chosen, such as interviews, visual inspection, structured observation and water-cooler discussions. Water-cooler discussions are conducted during coffee and lunch-breaks and make use of the fact that most information sharing within development companies are conducted during breaks [77]. This allows the researcher to get qualitative data that was current and truthful, but since the information often included the persons own opinions the information had to be verified to remove bias. Surveys were also used to provide quantitative results that would give depth to the research results. The surveys were deductive, hence based on previously elicited information, to verify the collected results and therefore conducted later in the company study. Different research methods are more appropriate to use depending on what sources are available, and therefore it is good to have a set of methods to choose from [69]. Creating a plan of what methods to use in what cells is suitable, but it should be considered a guideline that can be changed depending on the conditions of the study rather than a strict plan.

*Data collection and Data analysis and Alignment.* The first three steps of the process constitute the development of the design, whilst the following three steps of the process discuss how to execute the design. The fourth and fifth steps are defined as data collection and data analysis and alignment, and were executed incrementally, meaning that collected data was continuously analysed and aligned with previously collected information within the BAPO/PCF matrix, before new information was elicited and analyzed. Such analysis and alignment became essential since the data collection governed how the research would proceed, meaning that the collected information was used to evaluate if a certain research question had been answered or not. If the question had been answered the research could move on to a new cell,
2.3. THE BAPO/PCF FRAMEWORK

otherwise further time was spent to find more information.

Case studies require flexibility to be able to respond to different events that may occur during the course of the study, such as for instance sudden research opportunities. For instance during the company study an opportunity arose to do observation of the company’s system testing practices that would not arise again during the course of the planned study period. Because events, such as opportunities, have to be considered the research has to be flexible and can not follow a strict research plan. The BAPO/PCF framework does however support such flexibility, since the research questions connected to the BAPO/PCF matrix cells can be answered out of order, which allows the researcher to jump between cells and work with the question that is the most urgent at the moment. The progress of the research can also be measured through what research questions within the matrix have been answered. Each cell that gets filled with information, and its research question answered, constitutes a measurable part of the study progress, and once all questions in all cells have been answered the study is complete. The time spent with a certain matrix cell can either be planned according to a fixed time budget, or in a more ad hoc manner where the collected results drive the time spent working with a cell as well as the order which the matrix cells are traversed. There are academic papers that state that research using the BAPO perspectives should start with the business aspect because this has the largest impact on the company, followed by software architectures that has the second largest impact and so on until the organizational aspect has finally been analyzed [74]. No evidence could be found to support this claim during the company study, which used an approach where intermediate results decided what cell to jump to next in the study. The study at the company did not follow the matrix row by row, but instead company processes were first investigated, followed by software architectures, followed by company business and finally company organization. The BAPO/PCF matrix traversal was not chosen at random, it was chosen based on information that became available in the initial part of the forth process step and proved throughout the study to be valuable asset to keep the research focused. Even though several deviations were made from the research plan, because opportunities arose, it proved to be valuable to have a plan to fall back on, which is also supported by Runeson and Höst that state that a plan is crucial for a case study to succeed [69].

To strengthen the analysis of the collected information, in the company study, cause-effect chains (CEC) were combined with the BAPO/PCF matrix. These event chains link the research results together to provide a broader contextual view and also help to find the causes behind a result. Two different types of CEC’s were used during the company study, where the first type focuses on one path through the matrix, the simplest being from a certain perspective’s past to the same perspective’s current or from the current to the future, hence a chronological view of to the result. The first type of a CEC can also span between different company perspectives to provide a cross-perspective view, but always form a single path through the BAPO/PCF matrix that shows how an event in one perspective in time has affected another perspective in time and so on. The second CEC type differs from the first type in the sense that it starts in one matrix cell and spans out to several cells both in time and between perspectives, which shows the broader impact an event...
has had within the company. In the company study the CEC’s were developed post-elicitation of the research information by using inspection to find the internal connections between the results, which made it possible to draw clearer conclusions, develop predictions about the future of each company perspective, and also raise the result validity. Predictions for the future perspective of the matrix were developed using induction, with qualitative research data as input, based on the concept that if a specific action A in context B has outcome C in the past, which is repeated in the current in the same context B with result C, it would be likely that the same pattern would re-occur also in the future. A visual representation of the CEC’s was also developed that uses the research results, described in bullet-point lists in the matrix cells, which are connected by drawing arrows from the origin matrix cell to the cell(s) that have been impacted by the event in the origin cell. It should be noted that a CEC can never find a connection between the current to the past, or from the future to the current, but if there seems to be a connection backwards in time it might be possible to create another CEC that was previously overlooked. An example of how to visualize the second type of a CEC is presented in Figure 2.4, which shows how the introduction of unit-testing impacted the studied company in a larger context, which could have been overlooked if the scope of the research design had been narrower. The general reasoning behind CEC’s has also been visualized in Figure 2.5.

The BAPO/PCF matrix can also be used as a tool to visualize the research result quality. Several techniques were used to ensure data validity during the company study, but the primary technique was triangulation [69, 78], which states that in order for information to be valid it must be verified by at least three sources. These sources can be either artefacts or roles, for instance documentation, different roles at the company, etc, or results gathered by using several research methods, such as interviews, literature review, etc. The matrix can be used to show how triangulation was performed during a study by visualizing which sources were used to answer what research question within the BAPO/PCF matrix. This provides a graphical overview of the data va-
2.3. THE BAPO/PCF FRAMEWORK

Figure 2.4: Cause-effect chain example

Validation Discussion. The final step of the process was to validate the research results with the studied company. This was partially done in step four and five since those steps deal with information validation through the means of company review, but those reviews were on an information level, whilst the final validation was on a conclusion level to ensure that the conclusions drawn from the research information were reasonable and correct. The final validation can be done in different ways, like for instance writing a report that the company can review, or the results can be presented orally. In the company study the results were presented orally during a power-point presentation where the analytical matrix was once more used. All significant results from the study were broken down into bullet-points in the matrix cells and added to a power-point slide. The graphical representation of the results gave the audience a clear overview of the results, which could afterwards be discussed further in more detail. Errors or discrepancies in the results were finally analyzed further and rewritten in a final study report.
2.4 Discussion

The perceived advantages of applying the BAPO/PCF framework for case studies are related to the structure and the broadness of the multi-perspective view of the framework. By applying the multi-perspective approach to a case study it becomes possible to collect results specific to the different perspectives of the company, as well as in time. This provides a contextual structure that makes it clear where and how to acquire information to answer the research questions, but also how to find information that can link the results together. For instance by pointing out a role within the studied company that has knowledge about several perspectives, the information gained from this knowledge can be used to draw cross-perspective conclusions about the cause of a particular result. By looking at the entire matrix, including these cross-perspective results, longer chains of cause and effect can also be drawn that provides a deeper as well as broader understanding of results and of the company. Internal validity is also improved with this approach since as described by Wohlin et al. 2000 [58], a factor that is investigated because of its effects on another factor may itself be affected by a third factor, and if the third factor is overlooked there may be a threat to internal validity. Hence the BAPO/PCF framework improves the internal validity by giving the researcher the means of finding these connections between different factors that could have otherwise been overlooked with a narrower scope. It is however important to recognize that this framework, like other case studies, does not solve the issue regarding to what extent a factor within a given perspective affects another factor in another given perspective. A change within a company is seldom localized to a certain perspective of the company but rather has ripple effects to several different perspectives. An example from the company study is how changes to a company’s business goals resulted in the introduction of new processes, new architectural development methods as well as organizational change.

Other advantages that are side effects of the BAPO/PCF framework’s structure include support for project planning, visualization of the validation through triangulation, as well as to visualize the research results. The
framework supports research plans that are flexible and allow events, such as research opportunities, to be taken advantage of as they arise during the study. Visualization of the triangulation provides an overview of the validation effort and also makes it easy for non-researchers to see how the results have been validated, for instance making it easier to trust results and conclusions that are previously unknown to the studied company. Result visualization has similar advantages in terms of providing an overview of the collected results, and help to find connections between the collected information.

As for general case studies this design does not limit what research methods that are applicable for acquiring information, but unlike many other case study designs that rely on purely qualitative methods for data elicitation and analysis, this design allows longitudinal design concepts to be used as well, which are quantitative. This is made possible by the chronological axis of the matrix that links the past to the future and therefore allows quantitative metrics to be developed. An example of such a metric could be organizational change over time, i.e. the company’s growth rate. Such information can also be used deductively to draw more plausible predictions about the future, for instance how the growth of the company will continue or decrease.

The design does provide context to the research results but by broadening the research it becomes necessary to sacrifice information depth in projects with fixed budgets. Hence one of the BAPO/PCF framework’s greatest strengths is also its primary weakness. There is a trade-off that must be made when using the framework, which states that in order to gain information depth in one perspective, information depth will have to be sacrificed in another perspective. The consequences of this trade-off has not been investigated in this study, and since the design has only been used during one empirical case this subject is still open to speculation. The broadness of the design does however make it more suitable for research with open ended questions where very little or nothing is known about the phenomenon under study, rather than research with narrow research questions of a more deductive nature.

The results and conclusions that were developed during the company study proved to have high validity, however since the design has only been used in one study it is uncertain if the same results would be achieved in another study. The design is flexible in many ways, including the core concept of the analytical matrix, but once again, since only the BAPO/PCF configuration has been tested nothing can be said about another configuration even though it can be speculated that another configuration could be more beneficial in another company context.

2.5 Conclusions

This paper introduced the BAPO/PCF framework for structuring case studies in software engineering and ensuring they cover multiple perspectives. The framework was evaluated in a case study of verification and validation practices in a Swedish software company.

The combination of the BAPO (Business, Architecture/technical, Process and Organization) and PCF (Past, Current and Future) dimensions resulted in 12 sub areas to consider in designing and executing the case study. The matrix
allowed the research questions to be connected to the research effort, as well as providing the researcher with tools for result visualization, project planning and result validation. The structure also allowed an analysis of cause-effect chains across perspectives and in time providing a broader understanding and increased validity. Most importantly the framework helped uncover issues and connections the company themselves were not aware of.

The largest perceived disadvantage is that the approach can become too broad and therefore require considerable effort to cover all 12 sub areas. This can be addressed by sacrificing depth of analysis within less prioritized sub areas but this needs further research.

In summary, a flexible yet powerful case study research design can be created by adding structure through the use of an analytical matrix and a simple process to adapt it to the context being studied. The analytical matrix used, based on BAPO and PCF, can be of general value for such software engineering research.
Chapter 3

Paper B


E. Börjesson, R. Feldt

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.
3.1 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 [79], especially since software is prone to changing requirements, maintenance, refactoring, etc., which requires extensive regression testing. Regression testing should be conducted with configurable frequency [80], 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 [81, 82]. 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 [83] 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 [10, 34], Behavioral Driven Development [84], 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 [14, 15]. This uncertainty has resulted in the development of automated test techniques explicit for system and acceptance tests, e.g. Record and Replay (R&R) [5, 12, 85]. 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 [86], e.g. API, code, or GUI layout change, which in the worst case can render entire automated test suites inept [53]. 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 [87]. 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 Commercial-Tool\(^1\), and one open source, called Sikuli \cite{87}, in an industrial context to answer the following research questions:

[A] Is visual GUI testing applicable in an industrial context to automate manual high-level system regression tests?

[B] 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 3.2 presents related work followed by section 3.3 that describes the case study design. Section 4.4 presents results which are then discussed in section 6.5. Section 6.6 concludes the paper.

### 3.2 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 \cite{19} 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 \cite{54}. 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. \cite{12} 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) \cite{5,12,85}. R&R is based on a two step process where user mouse and

\(^1\)For reasons of confidentiality we cannot disclose the name of the tool.
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 [53] 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 [87], 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) [88, 89]. 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 [50].

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 [12], 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 [80] and has therefore been subject to academic research. The academic research has resulted in both tools and frameworks for acceptance test automation, including tools for GUI-interaction [32], but to the authors’
knowledge there is no research using visual GUI testing for acceptance testing.

3.3 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 3.1, 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 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.

### 3.3.1 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. [90], 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, chosen...
3.3. CASE STUDY DESCRIPTION

![Diagram of experimental setup]

Figure 3.2: Visualization of the experimental setup.

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) [91], 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 3.2.

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 [92]. 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.

### 3.3.2 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 3.3. VNC is scalable for distributed systems so the level of complexity of the industrial test system setup, Figure 3.3, was directly comparable to the complexity of the experimental setup used during the pre-study, Figure 3.2.

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 automated test cases, the most complex properties have been listed below:

[A] The number of physical computers in the subsystem the test case required access to.

[B] Which of the available simulators for the subsystem the test case required access to.

[C] 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 3.1 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...
CHAPTER 3. PAPER B

<table>
<thead>
<tr>
<th>Test case</th>
<th>Physical computers</th>
<th>Run-time config.</th>
<th>Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>2</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>Test case 2</td>
<td>2</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>Test case 3</td>
<td>2</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>Test case 4</td>
<td>2</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>Test case 5</td>
<td>3</td>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>

Table 3.1: 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.

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.

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.

3.4 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.

3.4.1 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 6.1 that shows which property had impact on what tool quality criteria defined by Illes et al. [90], described in section 3.3. 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.** CommercialTool 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 3.3, 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
<table>
<thead>
<tr>
<th>Property</th>
<th>CommercialTool</th>
<th>Sikuli</th>
<th>Impacts</th>
<th>Favored tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed in</td>
<td>C#</td>
<td>Jython</td>
<td>F/P/VS</td>
<td>S</td>
</tr>
<tr>
<td>Script language syntax</td>
<td>Custom</td>
<td>Python</td>
<td>F/U/M</td>
<td>S</td>
</tr>
<tr>
<td>Supports imports</td>
<td>No</td>
<td>Java and Python</td>
<td>F/U/E/VS</td>
<td>S</td>
</tr>
<tr>
<td>Image representation in tool IDE</td>
<td>Text-Strings</td>
<td>Images</td>
<td>F/U/M/P</td>
<td>S</td>
</tr>
<tr>
<td>Real-time script execution feedback</td>
<td>Yes</td>
<td>No</td>
<td>U/M</td>
<td>CT</td>
</tr>
<tr>
<td>Image recognition sweeps per second</td>
<td>7</td>
<td>5</td>
<td>F/R/U</td>
<td>CT</td>
</tr>
<tr>
<td>Image recognition failure mitigation</td>
<td>Multiple algorithms to choose from</td>
<td>Image similarity configuration</td>
<td>F/R/U/E/M/P</td>
<td>CT</td>
</tr>
<tr>
<td>Test suite support</td>
<td>Yes</td>
<td>Unit tests only</td>
<td>F/U/M/P</td>
<td>-</td>
</tr>
<tr>
<td>Remote SUT connection support</td>
<td>Yes</td>
<td>No</td>
<td>F/U/P</td>
<td>-</td>
</tr>
<tr>
<td>Remote SUT connection requirement</td>
<td>Yes</td>
<td>No</td>
<td>F/U/P</td>
<td>S</td>
</tr>
<tr>
<td>Cost</td>
<td>10.000 Euros per license per computer</td>
<td>Free</td>
<td>U/LP</td>
<td>S</td>
</tr>
<tr>
<td>Backwards compatibility</td>
<td>Guaranteed</td>
<td>Uncertain</td>
<td>F/M/GVQ</td>
<td>CT</td>
</tr>
</tbody>
</table>

Table 3.2: 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
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.

*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 3.3, 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 CommercialTool since some companies may 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 3.3 and their results are summarized in Table 3.3. 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
Table 3.3: Academic experiment results. CT stands for CommercialTool. Type indicates if the experiment was non-animated or not and Desc. describes the experiment.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Type</th>
<th>Desc.</th>
<th>CT success rate (%)</th>
<th>Sikuli success rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>non-animated</td>
<td>Calculator</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>non-animated</td>
<td>Icon finder</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>animated</td>
<td>Car Finder</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>animated</td>
<td>Radar trace</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

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.

### 3.4.2 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 3.3, 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 3.3, 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 3.4 together with the script development time and number of steps in the corresponding manual test case scenario.

Table 3.4 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 3.3, to allow the collected data to be used for estimation. Estimation based on the average execution times, from Table 3.4, 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 3.4 the script development time for the scripts, taken from Table 3.4, 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
Table 3.4: Metrics collected during test case automation. CT stands for Commercial Tool, ATC for automated test case and TC steps for the number of test steps in the scenario of the manual test case.

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

Figure 3.4: Boxplot showing development time of the five scripts in each tool.

The data in Table 3.4 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 3.4, 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.

### 3.5 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 6.1, 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 GUls and that Sikuli also had good success-rate on animated GUls as well. Hence, this study shows that visual GUI testing works for tests on non-animated GUls and perceivably also for animated GUls. 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 [32, 85]. 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 unexplained 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.
3.6 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.
Chapter 4

Paper C

Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study

E. Alégroth, R. Feldt, H. H. Olsson

Accepted at the 6th International Conference on Software Testing Verification and Validation (ICST’2013), Luxembour, March 18-22, 2013.
Abstract

Visual GUI testing (VGT) is an emerging technique that provides software companies with the capability to automate previously time-consuming, tedious, and fault prone manual system and acceptance tests. Previous work on VGT has shown that the technique is industrially applicable, but has not addressed the real-world applicability of the technique when used by practitioners on industrial grade systems. This paper presents a case study performed during an industrial project with the goal to transition from manual to automated system testing using VGT. Results of the study show that the VGT transition was successful and that VGT could be applied in the industrial context when performed by practitioners but that there were several problems that first had to be solved, e.g. testing of a distributed system, tool volatility. These problems and solutions have been presented together with qualitative, and quantitative, data about the benefits of the technique compared to manual testing, e.g. greatly improved execution speed, feasible transition and maintenance costs, improved bug finding ability. The study thereby provides valuable, and previously missing, contributions about VGT to both practitioners and researchers.
4.1 Introduction

To date, there are no industrial case studies, from the trenches, that visual GUI testing (VGT) works in industry when used by practitioners, nor data to support the long-term viability of the technique. In our previous work, we have shown that VGT is applicable in industry, even for testing of safety-critical software [17]. However, previous work has been essentially driven by researchers, e.g. they applied VGT techniques, compared the resulting test cases to earlier manual efforts, and then collected feedback and refinements from the industrial practitioners. There is a risk that this type of research does not consider all the complexities and problems seen by practitioners when actually applying a technique in practice. Furthermore, researcher driven studies are often smaller in scale and cannot evaluate longer term effects such as maintenance and refactoring of the test scripts or effects on, and of, changes to the system under test (SUT). Hence, there is still a gap in VGT’s body of knowledge regarding if the technique is applicable when performed by industrial practitioners in a real world development context.

In this paper we aim to bridge this gap by presenting an industrial case study from a successful project, driven entirely by industrial practitioners, with the goal to transition into VGT at the company Saab AB, subdivision security and defense solutions (SDS). The company chose VGT because of its ability to automate high system-level test cases, which previous automation techniques, e.g. unit testing [10, 34] and record and replay (R&R) [5, 12, 85], have had shortcomings in their ability to achieve. High system-level tests developed with automated unit tests have become both costly and complex, thereby spurring a discussion if the technique is applicable for anything but the low system-level testing, for which it was developed [14]. Furthermore, R&R techniques, which were developed for automation of system-level tests, are instead limited by being fragile to GUI layout and API change. Limitations that in the worst case have caused entire automated test suites to become inept [53]. Hence, the previous techniques have shortcomings in terms of flexibility, simplicity and robustness to make them long-term viable.

However, in this case study we show that VGT can overcome these limitations. Hence, showing that VGT has the capability to automate and perform industrial grade test cases that previously had to be performed manually, with equal or even greater fault finding ability, at lower cost. Capability provided by the technique’s use of image recognition that, in combination with scenario based scripts, allow VGT tools to interact with any graphical object shown on the computer monitor, i.e. allowing VGT scripts to emulate a human user. In addition, the study presents the practitioners’ views on using the technique, e.g. benefits, problems and limitations, when performed with the open source tool Sikuli [87]. Consequently, this work shows that VGT works for testing of real-world systems when performed by practitioners facing real-world challenges such as refactoring and maintenance of the SUT. The specific contributions of this work therefore include,

[A] An account on how the transition to VGT was successfully conducted by industrial practitioners for a real-world system.

[B] The industrial practitioners experiences and perception on the use of
4.2 Related Work

The concepts of using image recognition for GUI interaction is quite old and has been evaluated in a considerable body of knowledge. Work on using image recognition for GUI automation can be traced back to the early 90s, e.g. Potter [19] and his computer macro development tool, Triggers. Other early work in this area include the work of Zettlemoyer and Amant that used image recognition in their tool VisMap, which was used to automate the interaction with a visual scripting program as well as the game Solitaire [54]. However, this work focused on using image recognition for automation which we differentiate from testing since not all tools developed for GUI automation are intended for testing and vice versa.

The body of knowledge on using GUI interaction for testing is also considerable, e.g. shown by Adamoli et al. [12] in their paper on automated performance testing that covers 50 papers on automated GUI testing. Automated GUI testing can be performed with different techniques but the most common approach is referred to as record and replay (R&R) [5,12,85]. R&R consists of two steps. First a recording step where user input, e.g. mouse and keyboard interaction, to the system under test (SUT) is recorded in a script. In the second step, the recorded script can automatically be replayed for regression testing purposes. Different R&R tools record SUT interaction on different levels of GUI abstraction where the most common are on GUI bitmap level, i.e. using coordinates, or GUI widget level, i.e. using software references to buttons, textfields, etc. However, both approaches suffer from limitations that affect their robustness. Coordinate based R&R has the limitation that it is sensitive to GUI layout change whilst being robust to SUT code change. Widget based R&R, in contrast, is sensitive to SUT API or code structure change [53], but is instead robust to GUI layout change.

Image recognition based GUI testing with scenario based scripts, which we refer to as visual GUI testing (VGT), does not suffer from these limitations but it is only recently that the technique started to emerge in industry. One plausible explanation to this phenomenon is that the image recognition is performance intensive and it is not until now that the hardware has become powerful enough to cope with the performance requirements. VGT is a tool-supported technique, e.g. by Sikuli [87], EggPlant, etc., which conducts testing...
through the top GUI bitmap level of a SUT, i.e. the actual bitmap graphics shown to the human user on a computer monitor. Hence, scenario based VGT scripts can emulate a human user and can therefore also test all applications, regardless of implementation or platform, e.g. web, desktop, mobile. In most VGT tools the scenarios have to be developed manually, but there are also tools, e.g. JAutomate, which has record and replay functionality. Typical VGT scripts are executed by first providing the SUT with input, i.e. clicks or keyboard input, after which the new state of the system is observed, using image recognition, and compared to some expected output, followed by a new sequence of inputs, etc. In contrast to previous GUI testing techniques, VGT is impervious to GUI layout change, API or even code changes. However, VGT is instead sensitive to GUI graphics changes, e.g. changes in graphics size, shape or color.

Another approach to GUI testing is to use models, e.g. using finite state machines to generate test cases [88, 89]. These models generally have to be constructed manually, but automatic approaches, e.g. GUI ripping proposed by Memon [50], also exist. The benefit with GUI ripping is that it mitigates the extensive costs related to model creation. Costs that originate in the complexities of developing a suitable model. The limitation of this approach is that it is dependent on the SUT implementation, e.g. development language.

The area of GUI interaction based testing and automation is therefore quite broad but still limited in regards of empirical studies in real-world contexts with industrial grade software systems. R&R tools have been compared [12] and evaluated in industry, for both system- and acceptance-test automation, but, to our best knowledge, it is only our own work that evaluates VGT in an industrial context [17]. Our previous work is however limited since it was conducted only for a small set of real-world test cases and since the VGT automation was performed by researchers rather than practitioners. Hence, the body of knowledge on VGT, to the authors best knowledge, lacks industrial case studies that report on the real-world use of the technique.

Most research on GUI based testing focuses on system testing. However, acceptance testing is an equally important, valid and plausible test aspect to consider, i.e. tests where requirements conformity is validated through end user scenarios performed regularly on the SUT [80]. Scenario based acceptance tests do however distinguish themselves from system tests by including more end user specific interaction information, i.e. how the system will be used in the end users’ domain. Automated acceptance testing has also been a subject of much research, which has resulted in both frameworks and tools, including research into GUI interaction tools [32]. However, to the authors’ best knowledge, only our previous work has considered the subject of using VGT for acceptance testing.

4.3 Research methodology

This section will present the company where the VGT transition was performed as well as the research methodology used to collect data during the case study.
4.3. Research Methodology

The case study presented in this paper was conducted in collaboration with, and at, the Swedish company Saab AB, subdivision SDS, in the continuation of this paper referred to as Saab. The study was conducted at the company because they had taken the initial steps towards transitioning into VGT to automate their current manual testing, which presented an opportunity to collect data to bridge the current gap regarding VGT’s real world applicability. Figure 4.1 visualizes the stages of the case study, which will be presented in more detail in the following section based on the guidelines for reporting case studies presented by Runeson and Höst, 2009 [69].

Saab develops military control systems for the Swedish military service provider on behalf of the Swedish military forces. The system is, when deployed in the field, distributed between several mobile nodes and provides the ability to map the position of friendly and hostile forces on the battlefield and share this information among the nodes. Hence, the core functionality of the system relies on a map visualization, provided by a map engine, which allows the user to place symbols representing military units onto the map. Due to the system’s intended use it is considered both safety and mission critical. In addition, the system is developed for a touchscreen monitor for use while the node is in motion, i.e. buttons and other graphical GUI objects are larger than a conventional desktop application to mitigate faulty system interaction when used in rough terrain. The system is both developed and maintained by the company, with a development team that is independent from the testing team. In addition, the system has a very large and complex requirements specification aligned with 40 test specifications built from roughly 4000 use cases which has an estimated manual execution time of 60 man-weeks (2400 man-hours).
4.3.2 Research process

The case study consisted of three stages, shown in the leftmost column (named ‘Stage’) in Figure 4.1. The first stage was explorative in nature, the second sought to improve and support the VGT transition and the third was descriptive in nature. In the first stage, the row named ‘Pre-study’ in Figure 4.1, a workshop was conducted with the goal of collecting information about the company’s goals with the VGT transition, their manual test practices, the SUT, etc. This information was collected using unstructured open interviews with the testers that were driving the VGT transition at the company. Unstructured open interviews were chosen because very little was known about the company at this stage of the study. In addition, several documents were acquired that could provide further information about the manual test suite and the SUT.

In the second stage of the case study, which was four calendar months, a communication process was followed to allow the testers driving the VGT transition and the research team to exchange information on a regular basis, i.e. the row named ‘Case study’ in Figure 4.1. The communication process was put in place for two reasons. First because the project was to be driven by the testers at the company rather than the research team; the latter deliberately distanced themselves from the project in order for all collected data to genuinely portray VGT’s use in the real world. The second reason was out of necessity due to the physical distance, i.e. 500 kilometers, between the research team’s location and the company. The information exchange took place more often at the start of the project, at least once each week, since the research team had deeper understanding of VGT than the testers, i.e. the research team could provide the testers with expert support. This support included information of how to improve the VGT test suite that was being constructed but also suggestions of how to document the test suite and solutions to specific, low-level, problems that the testers had run into. In cases where the research team did not already have a feasible solution to a problem, the research team instead aided in the information acquisition to help the testers develop a solution. Further into the project, the information exchange became less frequent with telephone or mail communication roughly twice each month. During these discrete instances, challenges, limitations and solutions were discussed as well as the progress of the VGT transition. In addition, cost and time metrics were collected from the testers. Hence, the role of the research team in this stage of the project was two-fold. First to provide support for the VGT transition project, and second to acquire empirical data regarding the VGT transition from the testers.

In the third stage of the study, which aimed to portray the project and its outcome, a second workshop was held on site at the company, during which two structured deep interviews were held with the driving testers, shown in the row named ‘Post-study’ in Figure 4.1. Additionally, at this point of the project, an additional tester had joined the transition project who could provide a new perspective and further information about the transition and usage of VGT. The purpose of the interviews was to verify previously collected data, get a deeper understanding of the transition project as well as to collect further data on challenges, limitations and solutions that had been identified. Both
of the interviews were recorded and conducted using the same set of questions in order to raise the internal validity of the answers [69]. 71 questions were prepared for the interviews, 67 with the purpose of eliciting and validating previously collected information and 4 attitude questions aimed at capturing the testers views on VGT, post project. More specifically, the four questions were,

[A] Does VGT work? Yes/No, why?
[B] Is VGT an alternative or only a complement to manual testing?
[C] Which are the largest problems with VGT?
[D] What must be changed in the VGT tool, Sikuli, to make it more applicable?

In all of the questions, VGT refers to VGT performed with Sikuli [87], since Sikuli was the VGT tool that was used during the project. After the interviews, the recordings were transcribed in order to make the information more accessible. In addition, the answers were analyzed and compared among the respondents, i.e. the driving testers, to ensure that there were no inconsistencies in the factual data. The analysis showed that the respondents had answered the majority of the questions the same, including all attitude questions, but that they had complementing views on the attitude questions, i.e. what was the largest issue with working with VGT, etc.

4.4 Results and Analysis

The following section presents the results, and analysis of the results, divided according to the three stages of the VGT transition project, i.e. pre-transition (pre-study), during the transition (case-study) and post-transition (post-study) to VGT.

4.4.1 Pre-transition

The VGT transition at Saab was initiated out of necessity to shorten the time spent on manual testing. For each release, every six months to one year, the SUT went through extensive regression testing where a selected subset of the SUT’s test cases were manually performed. Each regression test session had a budget of four to six weeks of man-hours. The test cases were documented in 40 test suites, referred to as acceptance test descriptions (ATD). Each ATD consisted of a considerable set of use cases (UC), e.g. roughly 100, which each defined valid SUT input and the expected output. On a meta level these UCs were linked together into test chains that defined the test case scenarios, as exemplified in Figure 5.6. A test case was defined as a test path through a test chain that could be either linear, or contain branches, where a set of UCs, UC1 and UC2 (Top left of Figure 5.6), were first executed to set up the SUT in a specific state. The set up was then followed by the execution of one of a set of optional UCs, UC3A-C (Middle of Figure 5.6) to create a test path. Test paths could also have varying length, as exemplified in the figure where UC3A (Middle left in Figure 5.6) is followed by UC3AA (Bottom of Figure 5.6) while
Figure 4.2: Example of a acceptance test description (ATD) test chain (to the left) constructed from a set of ATD use cases (to the right). In the example the test chain contains three unique test-paths, i.e. test cases, that were, prior to the VGT transition, executed manually. **UC** - Use case.

the other two branches (UC3B and UC3C) lack following UCs. Hence, each test chain could contained a set of branching test-paths, i.e. test cases, defined by either common or unique UCs. The modular architecture of the manual test cases provided a lot of flexibility but was also considered tedious since some test chains required a lot of setup while only performing a small/short test thereafter.

The manual test period, four to six weeks, for the SUT, was then followed by a factory acceptance test (FAT) with the customer, executed over an additional two to three weeks, to validate the system, i.e. six to ten weeks of testing in total. However, a FAT would only be initiated if the manual tests had been executed successfully. Hence, transitioning to VGT from manual testing would constitute a large gain for the company in terms of development time, cost and potentially raised quality, since a larger subset of test cases from the ATDs could be executed faster and at higher frequency [17]. Raising test frequency was also important since manual testing was the only means of testing the system, i.e. no other tests existed for regression testing purposes such as automated unit tests, etc.

Three VGT tools were evaluated for the project, i.e. EggPlant, Squish and Sikuli, to find one suitable for the VGT transition. A brief overview of the results of the evaluation is given in Table 4.1. The primary success factors during the evaluation, which took six man-weeks, were tool cost and script language ease of use. Each tool was evaluated based on its static properties as well as through ad hoc scripting and automation of actual use cases from the ATDs. In addition, the evaluation took into consideration the research teams’ previous work, i.e. comparison of different VGT tools [17].

The result of the evaluation was that EggPlant was a mature and suitable tool but that it was very expensive and that the tool’s scripting language was a limitation, i.e. it had a high learning curve and did not suit the modular design of the tests that the testers were aiming for. Squish, used by other departments at Saab, was not suitable either since it performed GUI based testing through manipulation of execution threads in the application. However, the SUT was
4.4. RESULTS AND ANALYSIS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>EggPlant</td>
<td>VNC support, Mature product, Powerful</td>
<td>High cost, Script language limitations</td>
</tr>
<tr>
<td>Squish</td>
<td>Reference based, fast</td>
<td>Limited thread based interaction, inability to work with the map</td>
</tr>
<tr>
<td>Sikuli</td>
<td>Open source (free), flexible, Python scripting language</td>
<td>Volatile IDE, lacks test suite support</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of advantages and disadvantages of the VGT tools evaluated during the VGT transition project.

running roughly 40 threads at a time, spread over different system components, which limited Squish ability to interact with the SUT. Additionally, the tool was unable to identify objects placed on the map, due to its limited image recognition capabilities, which was a key feature of the SUT that the VGT tool had to be able to cope with in order to be applicable. Lastly, Sikuli was evaluated and found to be a feasible option, partly because the tool is open source, and thereby carries no up front cost, but mostly because of the tool’s scripting language which is based on Python. Python was considered valuable since it has a familiar syntax, i.e. common to most imperative and object-oriented programming languages, and because Python provides the capabilities of an object-oriented programming language. The main limitation with Sikuli, that was identified at this stage of the project, was that the tool did not have built in support for either development or management of test suites. However, thanks to the power of the tool’s scripting language this was considered a minor obstacle since a custom solution could easily be developed by importing and extending existing testing and test suite libraries for Python. Another problem that was identified was that Sikuli did not have any built in virtual network connection (VNC) support, required to test the SUT’s distributed functionality. However, by pairing Sikuli with a third part VNC client-server application, this issue was also easily solved.

4.4.2 During transition

The VGT transition took place during roughly four calendar months, during which three representative ATDs were fully implemented into a VGT test suite. Representativeness was measured by the ATD’s complexity, where two of the chosen ATDs were considered more complex than the average 40 ATDs, whilst the third was equal in complexity to the remaining ATDs. The VGT test suite architecture, visualized in Figure 5.3, consisted of two main parts. First, a main script for each ATD that imported all the automated ATD test cases, i.e. the test chains built from use cases. The second part was the test cases themselves which were executed by the main script according to the numerous test paths in each test chain. This architecture was required since Sikuli does
not, as mentioned, provide any support for either development or management of test suites. The VGT test suite was also developed using external libraries, 'lib' in Figure 5.3. One of these libraries was a Python library for formatting and producing output. Output that could be viewed graphically through any web browser, i.e. the result of each test case was visualized as passed or failed in a table. Additionally, a Java library for taking screenshots was incorporated in the VGT test suite. The screenshots provided additional value to the result output by capturing the state of the system when a bug was identified, i.e. the faulty state of the GUI was captured for further analysis and for manual recreation of the bug. According to the testers, this functionality made it easier to explain, and present, the faults they encountered to the developers, thereby quickening SUT maintenance time. In addition, all global variables used in the scripts were placed in its own library called 'glob', whilst the 'cfg' library included all external paths, i.e. paths to where to save log files, find the external libraries, etc.

After automation of the three ATDs, the testers compared the VGT test suite’s execution time against the manual test suite execution time. Results showed an estimated speed up of a factor 16, from two work days (16 hours) to 1 hour for the two complex ATDs and from 1 day to 30 minutes for the third. Hence, the automation constituted a huge gain in test case execution time with no reported detrimental effects on bug finding ability, i.e. all bugs in the system that were identified using manual test practices could also be identified using the VGT test suite. In addition, due to the quicker execution speed, the automated ATDs could be run several times in sequence. The iterative test suite execution placed the SUT in states that the manual test cases did not cover. Consequently, three new faults were uncovered that had not been identified earlier with the manual testing. In addition, these bugs were automatically captured and recorded by the screenshot capabilities of the VGT test suite which made them simpler to present, recreate and motivate as faulty behavior to the developers. However, even with the much higher
execution speed, the testers reported that they were often asked, “Doesn’t it execute quicker than this?” The simple answer, as reported by one of the testers, is, “Sikuli, or VGT, is limited by the speed of the SUT”, i.e. the VGT test suite cannot run faster than the reaction speed of the SUT’s GUI. Consequently, the scripts often had to be slowed down, using delays, in order to synchronize them with SUT loading times to ensure that the SUT’s GUI was ready for new input before the script continued its execution.

During development, attempts were made to integrate the VGT test suite into the SUT’s build system, i.e. to allow completely automatic system regression testing after each new build. However, since the VGT test suite required manual setup and some configuration before execution, such a scheme was never implemented due to time constraints. Instead, the VGT test suite was run on an ad hoc basis, i.e. not periodically, but with much higher frequency than the previous manual testing. The higher frequency regression testing was reported as most beneficial for the development of the SUT since it provided the developers with quicker feedback.

4.4.2.1 VGT test suite maintenance for improvement

To ensure validity of the automated test scripts, they were developed as a 1-to-1 mapping of the manual test cases, i.e. the manual tests were used as a specification for the automated scripts. However, later during the project, the VGT test suite was subject to maintenance. The maintenance done to the test case scenarios included, but was not restricted to, modification of the order of script operations, in order to provide smoother and quicker test case execution, and further modularization to facilitate strategic reuse. Hence, breaking the 1-to-1 mapping in some of the test cases. However, the purpose of each automated test case, i.e. the functionality the test case aimed to verify in the SUT, was kept the same. Consequently, a conclusion can be drawn that strict automation, i.e. 1-to-1 mapping, of the manual test specification may not necessarily be the best automation approach. Rather, the specification should only be used to specify what to test in the SUT, not necessarily how. The reason is because with automatic testing you can, and often want, to improve the test execution speed as much as possible, which can be done by grouping certain actions together. In contrast, manual test scenarios need to be unambiguous and test actions defined logically to have high quality [93], which isn’t necessarily the fastest. Hence, the quality of a VGT script is greatly affected by how it is designed and implemented, i.e. narrowing the gap between testing and traditional software development.

The performed refactoring of the VGT test suite was required since this project was conducted under continuous time pressure, with project managers expecting quick results. This pressure resulted in, as presented by the testers, development of the first possible solution for certain problems which necessarily wasn’t always the best solution in terms of script quality, performance, reusability, etc. Additional refactoring was also required due to the testers inexperience of using Sikuli at the start of the project. Among the refactoring that was made, in order to improve maintenance of the scripts, all global variables were moved to a common namespace, i.e. ‘glob’, as shown in Figure 5.3. Hence, all variables were clustered in one library and then, together with
the libraries, ‘lib’ and ‘cfg’, imported to all scripts that required them.

During the VGT test suite maintenance, the testers observed that it was easier to maintain the scripts that they had written last since they had a clearer memory of what the scripts did. Additionally, they reported that whilst maintenance of their own code was almost as quick as writing code from scratch, maintenance of scripts written by the other tester took considerably longer. One solution to mitigate these problems would have been a common coding standard of how to name variables, write loops and branches, etc. This problem, once again, illustrates how VGT, using Sikuli, in many respects has more in common with traditional software development than testing. However, as reported by the testers and in contrast to traditional development, the maintenance work was made easier by the scenario based structure of the scripts and the intuitiveness provided by inclusion of images in the scripts, a feature provided by Sikuli’s IDE. It was perceived by the testers that pure Python code would have been more difficult to maintain; the in-script images simplified understanding and remembrance.

4.4.2.2 VGT test suite maintenance required due to SUT change

Three calendar months into the VGT transition project a huge change was made to the SUT which included replacement of the map engine. Since the map engine was part of the core functionality of the SUT this change also affected the VGT test suite, i.e. causing 85-90 percent of the scripts to fail and thereby require some kind of maintenance, which included changing 5-30 percent of the images in every maintained script. The maintenance effort required to get the VGT test suite working completely again took roughly three man-weeks (240 man-hours) of work, which is to be related to the VGT test suite development time of three man-months (1032 man-hours). Hence, the estimated maintenance time of the entire VGT test suite, all 100 percent of the test scripts, would be 25.8 percent of the development time, i.e. 266 man-hours, which can be compared to the manual test budget of 480 man-hours per SUT development iteration. Note, the 4-6 week manual execution time, 120 hours, is with two testers. Consequently, the estimated development time of all 40 ATDs would be 13760 man-hours (7.6 man-years) and assuming all of the tests broke, the maintenance time would be 3550 man-hours, equal to roughly 21 man-months of continuous work or equivalent to the budge of 7 iterations of manual testing, i.e. roughly 3.5 years. However, the time required to execute all of the 40 ATDs manually is estimated to 2400 hours. Hence, assuming that none of the tests required maintenance and the complete VGT test suite (40 ATDs) was executed continuously, i.e. 24 hours a day, the ROI for the entire development would be positive after roughly 8 days (199 hours), i.e. after executing all the 40 automated ATD’s 6 times. Additionally, for the three ATDs that were automated in the project, a positive ROI would be reached after 13 executions, i.e. after 32.5 hours of continuous execution, which is less than the time of the manual ATD execution, i.e. 80 man-hours.

These numbers, summarized in Table 5.7, do however not reflect the manual testing that is performed during the VGT test script development, required to validate test script conformance to the manual test specifications. Furthermore, the numbers do not take into account aspects such as the number of
4.4. RESULTS AND ANALYSIS

<table>
<thead>
<tr>
<th>Artefact</th>
<th>Dev. time</th>
<th>Maintenance of VGT test suite</th>
<th>Man. exe. time</th>
<th>Positive ROI reached after</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGT test suite (Project)</td>
<td>1032mh</td>
<td>266mh</td>
<td>80mh</td>
<td>13 VGT test suite executions</td>
</tr>
<tr>
<td>Entire test suite (Estimated)</td>
<td>13760mh</td>
<td>3550mh</td>
<td>2400mh</td>
<td>6 VGT test suite executions</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of development-, maintenance- and manual execution times (man-hours) and return on investment (ROI) (VGT test suite dev. time / manual exe. time) data acquired from the VGT transition project. mh - man-hour, h - hour

faults found during the test execution, i.e. quality gained from quicker feedback to the SUT developers and other benefits provided by the VGT test suite, e.g. identification of previously unknown bugs. With these aspects taken into account, the driving testers estimated that the currently achieved ROI of the VGT transition was neither positive or negative. Hence, their perception is that all future regression testing performed with the VGT test suite will provide positive ROI for the company. However, the numbers also show that it would be unfeasible to automate all the 40 ATDs since it would take 7.5 man-years. Hence, an important conclusion is therefore that a company may have to prioritize or be selective in which manual test suites they decide to automate. Furthermore, as described by the testers, VGT primarily solves cost and speed problems rather than raising quality. The higher test frequency can help identify bugs faster, but bugs are only found if covered by the test scenarios.

The testers encountered a set of additional problems during the VGT transition, which have been summarized in Table 4.3. The main problem was the volatility and instability of the VGT tool, i.e. Sikuli. Sikuli is still a release candidate, i.e. not a finished product, and therefore suffers from some lingering bugs. These bugs affect the stability of the tool’s IDE that is prone to failure in certain instances, e.g. if the execution thread of a script is manually terminated, or if the tool is terminated with an unsaved script, etc. The solution to solve these problems has been to only use Sikuli’s IDE for script development and instead run the developed VGT test suite from the command line, which was found to greatly improve stability.

The single largest problem, as described by the developers, was however the failure rate of Sikuli’s image recognition algorithm, which was not improved by running the scripts from the command line. Estimates done by the testers indicate that the VGT test suite only had a success rate of 70 percent. This low success rate has been established by the testers to be due to the use of
VNC. The VNC server-viewer application is used to run test cases that are distributed over several physical computers. However, not all of the test cases require the VNC connection and when these tests were executed against the SUT, without VNC, the testers observed a close to 100 percent success rate, even when the VGT test suite was left to its own devices for over 24 hours. Consequently, the solution that was employed, during the pre-transition stage of the project, to allow Sikuli to test the distributed system, also proved to be the largest problem for the stability of the scripts. The cause of the problem has not yet been verified but the hypothesis is that the problem is related to network latency, causing the remote images sent from the VNC server to the VNC viewer to be distorted, causing the image recognition algorithm to fail.

Additional problems caused by the VNC solution relates to the mouse pointer. Sikuli, when executed locally, disregards the mouse pointer, i.e. removes it from the screen, when it’s performing the image recognition. However, when executed over VNC the mouse pointer cannot be removed and if placed in the wrong position, e.g. in front of the sought button, it causes the image recognition to fail. The problem can easily be mitigated by adding operations in the script to continuously move the mouse pointer to a safe location. However, this solution is inconvenient and adds unnecessary code and execution time to the scripts. Additionally, as reported by the testers, it adds frustration to the script development.

Yet, even though there were many problems, challenges and limitations that hindered the VGT transition, the testers still claim that they had not encountered anything that they could not automate using Sikuli. Additionally, the testers experienced that the development itself contributed to raising the quality of the SUT since it required them to perform the test cases manually several times to obtain a greater knowledge of how to automate them. Hence, the development work itself helped uncover several faults in the SUT. Faults that could later also be identified automatically by the VGT test suite.

4.4.3 Post-transition

After the VGT transition was completed, a second workshop was held on site at the company during which structured interviews were performed with the testers driving the project. The purpose of the interviews was primarily to verify previously collected information but also to capture the testers views on if VGT is viable for system- and acceptance-testing in industry.

During the interviews, four attitude questions were asked, presented in Section 4.3 and summarized in Table 4.4. For the first question, does VGT work, the interviewees were clear that it did. Two motivations stated by one of the testers was, “It is such a good way to quickly run through and make sure that everything still works and you can use it on any system”. An additional motivation from another tester was, “VGT is the only thing that works on our system”. Hence, VGT is perceived not to be bound to any specific implementation language, API, etc., and its image recognition capabilities therefore allows it not only to interact with one application at a time, but seamlessly interact with different applications at once.

For the second question, when asked if VGT is a complement or a replacement for manual testing, the testers stated that it is a complement, “It’s part of
## 4.4. Results and Analysis

<table>
<thead>
<tr>
<th>Title</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>VNC</td>
<td>VNC has negative effects on the image recognitions ability to identify GUI graphics</td>
<td>Minimize use of VNC if possible, use high-quality VNC application, use Egg-Plant</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Understanding other developers scripts can be problematic even with the scenario based structure of the scripts</td>
<td>Enforce coding standards to raise understandability and readability of the scripts</td>
</tr>
<tr>
<td>1-to-1 mapping</td>
<td>1-to-1 mapping between manual and automated tests is not always possible or favorable</td>
<td>Modularization of test scripts can increase test execution speed and reusability. Hence, a 1-to-1 mapping should be strived for only if it does not have detrimental effects on test quality.</td>
</tr>
<tr>
<td>Sikuli IDE volatility</td>
<td>Sikuli is not a finished product and therefore cause the Sikuli IDE to fail unexpectedly</td>
<td>Use IDE only for script development but execute scripts from command-line</td>
</tr>
<tr>
<td>Lack of documentation</td>
<td>Sikuli’s API is poorly documented</td>
<td>Ensure internet connectivity to make it possible to look up solutions and other information online.</td>
</tr>
<tr>
<td>Image recognition</td>
<td>Many problems were identified with Sikuli’s image recognition, e.g. spontaneous inability to find images, click operations performed next to intended location, etc.</td>
<td>No one solution was identified, but potential solutions include fine-tuning the scripts, better selection of images, running scripts locally without VNC, etc.</td>
</tr>
</tbody>
</table>

Table 4.3: Summary of problems and solutions identified during the VGT transition project.
<table>
<thead>
<tr>
<th>Nr</th>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Does VGT work? Yes/No, why?</td>
<td>Yes, only technique the testers have identified capable of automating their manual tests.</td>
</tr>
<tr>
<td>2</td>
<td>Is VGT an alternative or only a complement to manual testing?</td>
<td>complement, since it can only find faults covered by the scripted scenarios.</td>
</tr>
<tr>
<td>3</td>
<td>Which is the largest problem with VGT?</td>
<td>The volatility of the tool and the image recognition.</td>
</tr>
<tr>
<td>4</td>
<td>What must be changed in the VGT tool, Sikuli, to make it more applicable?</td>
<td>Support for testing of distributed systems, e.g. through VNC.</td>
</tr>
</tbody>
</table>

Table 4.4: Summary of the driving testers’ responses to the four attitude questions asked during the second workshop.

*the test palette*. Based on their perception, VGT may work as a replacement for smaller systems, but for large and complex systems it is neither suitable or plausible that this could be achieved. The reason is because it is improbable that test scenarios can be devised that cover all states of a large systems, which is equally unlikely for manual scenario based test cases. Instead, manual exploratory testing should be used to uncover new faults.

For the third question, what is the biggest problem with VGT, one of the testers stated, “I don’t see any problems with it, but we need to get around the fact that it does not always work and that we always don’t know why.”, referring to Sikuli’s volatility. Another tester answered, “The image recognition comes with an inherent uncertainty”, i.e. fragility to unexpected SUT behavior, etc. However, the testers had a pragmatic approach to these issues and stated, “Sikuli is a program, it’s also a system and systems have faults”. Hence, they had accepted the tools limitations but also identified that most of these limitations could be mitigated through structured script development, redundancies in the scripts and other failure mitigation practices.

Finally, when asked what can be improved with the VGT tool, the testers answered that the reliability of the tool should be increased or at least a study should be conducted that can explain why the image recognition works in some cases, for some images, and not for others. Additionally, the tool documentation needs to be improved and since one of the largest issues during the VGT transition was found to be how the tool interacted with VNC, Sikuli should be fitted with VNC capability, similar to EggPlant. As stated by the developers, “EggPlant was much more stable with VNC. We have not managed to make Sikuli as stable.”.

Due to the success of the transition project, i.e. identification of previously unknown faults in the SUT and the perceived cost-effectiveness of the tech-
4.5 DISCUSSION

The data collected during the industrial case study shows that the transition to VGT was both successful and of benefit to Saab, benefits summarized in Table 4.5. Firstly, the execution speed of the company’s previously manual tests was greatly improved that allows for greater test frequency and thereby faster feedback to the developers, i.e. from months to hours. Secondly, and perhaps more importantly, the automated tests did not just identify all the faults found by the manual tests, but also previously unknown faults. Consequently, this report provides support that VGT does not just lower testing costs, but can also help raise software quality. However, as also reported, the transition cost of several large manual test suites can be extensive, so a cost-benefit prioritization model of what test suites to automate should be developed, which is a subject of future work. Thirdly, the return on investment (ROI) of transitioning to the technique, i.e. automating the manual tests, was perceived by the driving testers to become positive after only one iteration of SUT development. A claim supported by our previous research [17], which came to the same conclusion at another Saab company. Additional support comes from the fact that the manual tests are continuously performed during VGT transition to ensure script validity, i.e. not taking time away from the normal manual testing, and the benefit of faster fault identification due to raised test frequency. Manual testing cost increases linearly with each development iteration, but VGT only has an initial cost for developing the automated test suites after which the cost of executing the scripts is constant. Hence, due to the execution speed of a VGT test suite, the number of executions required to reach a positive ROI can be performed quickly, as shown in Table 5.7.
<table>
<thead>
<tr>
<th>Description</th>
<th>Past</th>
<th>Current</th>
<th>Benefit (versus manual testing) or improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATD execution time</td>
<td>1-2 days per ATD (60 man-weeks for all ATDs), manually</td>
<td>0.5-1 hour per auto. ATD (Estimated 33 hours for all automated ATDs)</td>
<td>Test execution 16 times faster, higher test frequency, quicker feedback to developers</td>
</tr>
<tr>
<td>Fault finding</td>
<td>-</td>
<td>3, previously unknown, faults found</td>
<td>VGT provides greater fault identification ability, higher system quality</td>
</tr>
<tr>
<td>Test ROI</td>
<td>Linear cost (Manually)</td>
<td>Constant cost after 1 iteration (Automatic)</td>
<td>Positive ROI after one iteration, feasible development cost</td>
</tr>
<tr>
<td>Script maintenance cost</td>
<td>Unfeasible in the worst case for previous GUI test techniques</td>
<td>~25% of the development cost of the VGT test suite (Saab project, with Sikuli)</td>
<td>Maintenance cost perceived feasible</td>
</tr>
<tr>
<td>Sikuli executed over VNC</td>
<td>~70% success rate with VNC</td>
<td>100% success rate without VNC</td>
<td>Sikuli stable when executed locally</td>
</tr>
</tbody>
</table>

Table 4.5: Summary of quantitative benefits and improvements identified during the VGT transition project.
4.5. DISCUSSION

Additionally, as shown in Table 4.5, other improvements were identified that are of benefit for future use of VGT and compared with previous GUI testing techniques, e.g. record and replay (R&R). Firstly, results show that the maintenance costs of a VGT test suite are not excessive, i.e. 25 percent of the development cost. In addition, the script refactoring was generally contained to parts of or specific scripts, which should be compared to the required maintenance of previous techniques, e.g. R&R, where entire test suites were rendered useless due to SUT change. Consequently, the black box nature of VGT, due to the image recognition, makes changes to the SUT maintainable. However, the collected data is not enough to draw a definitive conclusion that the maintenance costs of VGT scripts are feasible for industrial use; more research is needed on this in the future.

Secondly, Table 4.5 presents data regarding the stability of VGT when used together with a virtual network connection (VNC). VNC was used during the project because the SUT was distributed over several computers. However, this pairing was recognized as a large problem since it lowered the success rate of the automated test suite, when it should have succeeded, to roughly 70 percent, i.e. due to image recognition failures. The VNC problem was identified by running a subset of test scripts, which could be run locally, against the SUT that resulted in a success rate of 100 percent, even when the tests were rerun continuously for 24 hours. Hence, Sikuli’s image recognition was not the source of the problem, but rather it was the third party VNC application, mitigated by local VGT test script execution. However, since the system was distributed over several computers, i.e. nodes, this solution instead limited which test cases could be executed. Hence, this was not identified as a benefit but rather an improvement of how to use Sikuli to raise test suite stability. Consequently, either a better VNC application has to be obtained or VNC should be integrated into Sikuli as already available in the VGT tool EggPlant, which was perceived by the testers be much more stable in this regard. However, EggPlant, as reported by the testers, had other limitations, e.g. a high cost and, what they considered, an unintuitive and more restricting scripting language. Consequently, existing VGT tools suffer from important, but different limitations, that makes it likely that manual test execution will still have to complement automated testing. However, the testers’ common view is that VGT both works and provides substantial value to the company, even given the tools’ limitations.

The testers also identified other less quantifiable benefits with VGT during the project. One benefit being the techniques flexibility and ability to work with any application regardless of implementation language or even platform, i.e. web, desktop, mobile, etc. This flexibility allows VGT to interact with the SUT whilst also interacting with SUT related simulators, written in other programming languages, or even the operating system if required. This is a specific benefit of VGT that might or might not be present with other similar testing tools, such as R&R or GUI testing techniques that are specific to the GUI library in use. In addition, the VGT tool that was used, i.e. Sikuli, uses Python as a scripting language that provides the user with all of the properties of a lightweight, object-oriented programming language. These properties presents new interesting opportunities for automated testing but also new problems. Since the scripts follow the rules of traditional
software development they are also subject to the same types of faults, i.e. if implemented incorrectly they can contain bugs. Consequently, an inherent risk with complex scripts is that they report type 2 errors, i.e. false negatives, due to the scripts themselves being faulty. Hence, the question becomes, how do you verify the tests? Verification of scenario based scripts that strictly follow a manual test description can perceivably be done through comparison with the outcome of the manual tests. However, for more advanced VGT-based test applications a more complex verification technique might be required, e.g. based on oracles or properties, or other state-of-art techniques, to ensure that all faults in the SUT are identified.

4.5.1 Threats to validity

The main threat to the validity of this study is that it only presents results from one VGT transition project at one company. Hence, the results may have low external validity for other companies and domains [69]. In addition, since no structured data collection process could be performed by the driving testers during the project, due to resource constraints on the company’s end, quantitative metrics were only sparsely collected. The risk that very little quantitative data would be available from the project was identified already before the case study started and originates in the fact that this project was performed in a real-world context with real-world time constraints. Consequently, the results presented in this work are primarily based on data collected through interviews and are therefore mostly qualitative in nature. Further work is therefore required in more companies to provide additional support regarding the real-world applicability of VGT. Another threat is that the driving testers at the company might have been biased, i.e. wanting the transition to be successful. However, based on their thorough descriptions of faults, limitations and problems, this threat is considered minor.

4.6 Conclusion

In this paper we present an industrial case study from a successful visual GUI testing (VGT) transition project, performed by practitioners, at the company Saab AB, subdivision SDS. Additionally, problems, limitations and solutions that were identified during the project are presented. Furthermore, support is given that the maintenance costs of a VGT test suite, developed in Sikuli, are not excessive, i.e. in this project 25.8 percent of the VGT test suite development cost.

In previous work we have shown the industrial applicability of VGT, but in a smaller transitioning project driven by researchers with expert knowledge of the technique. The more extensive transitioning project presented in this paper was instead initiated from industry, and originated in the business need to shorten the execution time of manual regression testing. The main limitation of the VGT tool, Sikuli, used during the project, was its unpredictability, e.g. uncertain image recognition outcome and tool IDE instability, which was partly mitigated through local test suite execution via the command line. The benefits of VGT were reported to be the technique’s flexibility to work with any application, greatly improved test execution speed (16 times faster than
manual tests) and ability to identify all faults found by the previous manual tests. Furthermore, the VGT test suite could identify previously unknown faults, due to increased test execution speed that allowed the tests to be run several times in sequence. Results also showed that the VGT transition cost, of three automated acceptance test descriptions (ATD), was feasible, but that VGT transition of all of the company’s 40 ATDs would take 7.5 man-years of work, i.e. prioritization of the ATD transition will be required. However, the practitioners perception was still that the developed VGT test suite was beneficial and will provide the company with positive return on investment for all future use. Hence, even though there were problems and limitations, the practitioners’ perceptions, and collected data, show that VGT is a beneficial and feasible technique for industrial system test automation.
Chapter 5

Paper D

Visual GUI Testing in Practice: Challenges, Problems and Limitations

E. Alégroth, R. Feldt, L. Ryrholm

*In submission to the Empirical Software Engineering Journal.*
Abstract

In today’s software development industry, high-level tests such as system and acceptance tests are mostly performed with manual practices that are often costly, tedious and error prone. Test automation has been proposed to solve these problems but most automation techniques approach testing from a lower level of system abstraction. Their suitability for high-level tests has been questioned. High-level test automation techniques such as record and replay exist, but studies suggest that these techniques suffer from limitations, e.g. sensitivity to GUI layout or code changes, system implementation dependencies, etc.

Visual GUI Testing (VGT) is an emerging technique in industrial practice that aims to overcome many of the limitations experienced with previous test automation techniques. The core of VGT is image recognition applied to analyze and interact with the bitmap layer of a system’s front end. By coupling image recognition with test scripts, VGT tools can emulate end user behavior on almost any GUI-based system, regardless of implementation language, operating system or platform. However, VGT is not without its own challenges, problems and limitations (CPLs) but, like for many other automated test techniques, there is a lack of empirically grounded knowledge of these CPLs and how they impact industrial applicability. There is also a lack of information on the cost of applying this type of test automation in industry.

This paper reports an empirical, multi-unit case study performed at two Swedish companies that develop safety-critical software, while they transitioned their manual system test cases into tests automated with VGT. One complete test suite, consisting of 33 test cases, was automated in one of the projects and three test suites in the other, exact number of test cases is uncertain because of the structure of the manual test cases. The results from the study participants showed that the transitioned test cases could find defects in the tested systems and that no manual test cases were found that could not be automated. During these transition projects, a total of 58 different CPLs were identified and then categorized into 29 groups. These groups are presented and their implications for the transition to and use of VGT in industry is analyzed. In addition, four high-level practices are presented that were identified during the study, which would address about half of the identified CPLs. Furthermore, collected metrics on cost and return on investment of the VGT transition are reported together with information about the VGT suites’ defect finding ability. A total of nine defects were identified by the automated tests or during the transition, of which 5 were unknown to testers with extensive experience from using the manual test suites. The main conclusion from this study is that even though there are many challenges in transitioning to automated testing based on VGT, the technique is still valuable, flexible and considered cost-effective by the industrial practitioners.
5.1 Introduction

Currently available automation techniques for high-level testing, i.e. system and acceptance testing, leave more to be desired in terms of cost efficiency and flexibility, leaving a need for more empirical research into test automation [2, 3, 15, 16, 26, 53, 68, 95]. This need for automation support is one factor why manual testing is persistently used in industrial practice even though these practices are considered costly, tedious and error prone [1–6, 96].

Manual testing has long been the main approach for testing and quality assurance in the software industry and in recent years there has been a renewed interest in approaches such as exploratory testing that focus on the creativity and experience of the tester [94]. The more traditional manual tests are often pre-defined sets of steps performed on a high level of system abstraction to verify or validate system conformance to its requirement specification, i.e. system tests, or that the system behaves as expected in its intended domain, i.e. acceptance tests [28, 80–82]. However, software is prone to change and therefore requires regression testing [7, 8], which can lead to excessive costs since testers continuously need to re-test. Manually having to repeatedly follow the same test descriptions and look for the same problems is also tedious and error prone.

To mitigate these problems, whilst retaining or increasing the end quality of the software being tested, automated testing has been proposed as a key solution [6]. Many automated test techniques approach testing from lower levels of system abstraction, e.g. unit tests [10, 34, 35], which make them a powerful tool for finding component and function level faults. However, the use of these techniques for high-level testing has been questioned since empirical evidence suggests that high-level tests based on low-level test techniques become complex, costly and hard to maintain [2, 15].

Thus, a considerable body of work has been devoted to high-level test automation, resulting in techniques such as coordinate- and widget/component-based Record and Replay with a plethora of tools such as Selenium [97], JFCTestUnit [98], TestComplete [99], etc. The tools record the coordinates or properties of GUI-components during manual user interaction with the system’s GUI. These recordings can then be played back to emulate user interaction and assert system correctness automatically during regression testing. However, empirical studies has identified limitations with these techniques. They are typically sensitivity to GUI layout or code change and tools dependend on the specifics of the system implementation, etc., which negatively affect the techniques’ applicability and raises the cost of maintaining the tests [9, 15, 16, 53].

Visual GUI Testing is an emerging technique in industrial practice that combines scripting with image recognition in open source tools such as Sikuli [21], and commercial tools such as JAutomate [56]. Image recognition provides support for user emulated interaction with the bitmap components, e.g. images and buttons, shown to the user on the computer monitor and is therefore perceived to enable testing of any GUI-based system, regardless of implementation, operating system or even platform. Empirical research has shown the technique’s industrial applicability for high-level test automation with equal or even better fault finding ability than its manual counterparts [18]. However, VGT is not without its challenges, problems or limitations (CPLs), such as
CPLs related to tool immaturity and image recognition volatility. These CPLs have not been sufficiently explored in the existing studies and it is thus hard to give a balanced description of VGT to industrial practitioners as well as advice and help them in successfully applying it.

In this paper we present an empirical study performed at two different companies within the Swedish corporation Saab, in which two teams transitioned existing, manual system test cases to VGT test scripts. The two projects were independent from each other and in two different sites, one driven by a researcher and the other by industrial practitioners, yet both projects provided corroborating results.

During the study, a total of 58 CPLs were identified that were categorized into 29 groups of mutually exclusive CPLs that affect either the transition to, or usage of, VGT in industrial practice. These groups have implications for the industrial applicability of VGT, e.g. adding frustration and confusion during transition and usage, but also automated testing in general. Consequently, the results of this study add empirical evidence regarding CPLs, which is currently limited, to the general body of knowledge about automated testing [68]. In addition, four general solutions were identified during the study that mitigate or solve roughly half of the identified CPLs related to VGT. Solutions that provide support and guidance to industrial practitioners that intend to evaluate or use the technique. Furthermore, quantitative information was acquired during the study regarding the VGT suites’ defect finding ability, the VGT suites’ development costs and the projects’ evaluated return on investment. Information that provide decision support for industrial practitioners regarding the potential value and cost of transitioning their manual testing to VGT. Based on these results we draw a conclusion regarding the CPLs implications on the cost-effectiveness, value and applicability of VGT for high-level test automation in industry.

Hence, the specific contributions of this work are:

- Detailed descriptions of the challenges, problems and limitations (CPLs) that impact either the transition to, or usage of, VGT in industry.

- Descriptions of identified practices, from the industrial projects, which solve or mitigate CPLs that impact the transition to, or usage of, VGT in industry.

- Quantitative information on the defect finding ability of the developed VGT suites compared to the manual test suites that were used as specification for the automated tests.

- Detailed information on the cost of transition, usage and return on investment of the VGT suites, in context of the manual test suite execution cost.

The next section, Section 6.2, presents a background to manual testing and GUI-based testing as well as related work on previously used GUI-based test techniques and VGT. Sections 5.4 presents the results from the study, including identified VGT related CPLs, solutions to said CPLs, VGT bug finding ability, metrics on the cost of transitioning to VGT and ROI metrics. Section 6.5, presents a discussion regarding the results, future work, as well as the threats to validity of the study. Finally Section 7 will conclude the paper.


5.2 Background and Related work

The purpose of high-level testing, e.g. system and acceptance testing, is to verify and/or validate system conformance to a set of measurable objectives for the system, i.e. the system’s requirements [28]. These tests are generally performed on a higher-level of system abstraction since their goal is to test the system in its entirety, which also makes them hard to automate, generally requiring large and complex test cases. Tests that are particularly hard to automate are non-functional/quality requirements (NFR) conformance tests for NFRs such as usability, user experience, etc. The reason is because NFRs differ from the functional requirements since they encompass the system in its entirety, i.e. they depend on the properties of a larger subset, or even all, of the functional components. Hence, for a non-functional/quality requirement to be fulfilled, all, or most, of the components of the system have to adhere to that requirement, which is verified either during system or acceptance testing of the system. Both system and acceptance tests are, in general, based around scenarios [24, 49], but with the distinction that system tests only aim to verify the functionality of the system. In contrast, acceptance tests aim to validate the system based on end user scenarios, i.e. how the system will be used in its intended domain. Note that we use the word scenario loosely in this context, i.e. not just documented scenarios, e.g. use cases, but ad hoc scenarios as well. These tests should, according to [28, 80–82], be performed regularly on the system under test (SUT), preferably using automated testing. Automated testing is proposed because both manual system and acceptance testing are suggested to be costly, tedious and error-prone [1–6]. Therefore, a considerable amount of research has been devoted to high-level test automation techniques, which has resulted in both frameworks and tools, including graphical user interface (GUI) based interaction tools [32, 85].

GUI-based test automation has received a lot of academic attention, with research into both high-level functional requirement and NFR conformance testing, as shown by Adamoli et al. [12]. In their work on automated performance testing they identified 50 articles related to automated GUI testing using different techniques. One of the most common they identified was capture/record and replay (R&R) [5, 12, 85]. R&R is a two step approach, where user input, e.g. mouse and keyboard interaction performed on the SUT, are first captured in a recording, e.g. a script. In the second step, the recording is replayed to automatically interact with the SUT to assert correctness during regression testing. However, different R&R techniques capture recordings on different levels of system abstraction, i.e. from a GUI component level to the actual GUI bitmap level shown to the user on the computer’s monitor. The GUI bitmap level R&R techniques drive the test scenarios by replaying interactions at exact coordinates on the monitor, i.e. where the GUI interactions were performed by the user during recording. However, the assertions are generally performed on lower levels of system abstraction, i.e. component level, but some tools also support bitmap comparisons. In contrast, widget/component-based R&R techniques are performed completely on a GUI component level, i.e. by capturing properties of the GUI-components during recording and using these properties, in combination with direct interaction with the GUI components, e.g. invoking clicks, to perform interaction during
playback. However, both of these techniques suffer from different limitations that affect their robustness, usability, cost, but foremost their maintainability, suggested by empirical evidence related to the technique [3, 9, 16, 25–27]. The coordinate-based R&R techniques are sensitive to GUI layout change [5], e.g. changing the GUI layout will cause the script to fail, whilst being robust to changes in the code of the tested system. Widget/component-based R&R techniques are instead sensitive to API or code structure change, whilst being robust to GUI layout change [53]. In addition, the technique can pass test cases that a human user would fail, e.g. if a widget blocks another widget, the use of direct component interaction still allows the tool to interact with the hidden widget. Furthermore, because direct component interaction is used, the technique also requires access to the backend of the system and generally only work for SUTs written in one programming language. Some exceptions exist, e.g. TestComplete [52], which support many different programming languages, or even interaction with the Windows operating system, but not other operating systems, e.g. MacOS. Hence, even though previous techniques have properties that support their use for high-level testing, they still suffer from limitations that perceivably limit their use for systems written in different programming languages, distributed systems, and cloud based systems where access to the backend is limited. Thus, indicating that there is a need for more research into high-level test automation.

Visual GUI Testing is an emerging technique in industrial practice that uses tools with image recognition capabilities to interact with the bitmap layer of a system, i.e. what is shown to the user on the computer’s monitor. Several VGT tools are available to the market, including both open source, e.g. Sikuli [87], and commercial, e.g. JAutomate [56], but even so the technique is only sparsely used in industry. VGT is performed by either manual development, or recording, of scripts that define the intended interaction with the SUT, usually defined as scenarios, which also include images of the bitmap-components the tools should interact with, e.g. buttons. During script execution, the images are matched, using image recognition, against the SUT’s GUI and if a match is found an interaction is performed. This capability is also used to assert system correctness by comparing expected visual output with actual visual output from the SUT. Hence, VGT uses image recognition both to drive the script execution and assert correctness compared to previous techniques where image recognition was only used in some tools for assertions. Furthermore, VGT is a blackbox technique, meaning that it does not require any knowledge of the backend and can therefore interact with any system, regardless of implementation language or development platform, e.g. desktop (Windows, MacOS, Linux, etc.), mobile (iPhone, Android, etc.), web (Javascript, HTML, XML, etc.).

In our previous work [17], we performed an empirical, comparative, study with two VGT tools to identify initial support for the technique’s industrial applicability. The tools, Sikuli and CommercialTool\(^1\), were compared based on their static properties as well as their ability to automate industrial test cases for a safety-critical air traffic management system. 10 percent of the tested system’s manual test cases were automated with both tools, which showed that there was no statistical significant difference between the tools and that

\(^1\)The name of CommercialTool can not be disclosed due to confidentiality reasons.
both tools were fully capable of performing the automation with equal fault finding ability as the manual test cases. However, the study was performed by academic experts in VGT and therefore a second study was performed, driven by industrial practitioners in an industrial project [18]. Results of the second study showed that VGT is also applicable when performed in a real project environment when used by industrial practitioners under real-world time and cost constraints. Thus, providing further support that VGT is applicable in industry.

A general conception within the software engineering community is that automated testing is key to solving all of industry’s test-related problems. However, Rafi et al. [68] that performed a systematic literature review regarding the benefits and limitations of automated testing, as well as an industrial survey on the subject, found little support for this claim. In their work, they scanned 24,706 academic reports but only found 25 reports with empirical evidence of the benefits and limitations of automated testing. Additionally, they found that most empirical work focus on benefits of automation rather than the limitations. Furthermore, the survey they conducted showed that 80 percent of the industrial participants, 115 participants in total, were opposed to fully automating all testing. In addition, they found that the industrial practitioners experienced a lack of tool support for automated testing, e.g. the tools are not applicable in their context, have high learning curves, etc. Consequently, their work shows that there are gaps in the academic body of knowledge regarding empirical work focusing on the benefits and in particular the limitations of automated testing as well as the actual needs for test automation in industry. A gap that this work helps to bridge by explicitly reporting on the challenges, problems and limitations related to high-level test automation using VGT.

5.3 Industrial case study

The empirical study presented in this report consisted of two parts. First, a VGT transition project driven by a researcher at the company Saab AB, in the swedish city of Gothenburg, building on our previous work [17]. Second, a VGT transition project in another Saab company within the Saab corporation, in the swedish city of Järfälla, which was driven by industrial practitioners with minimal support by the research team. Hence, the two projects complement each other by providing information of academic rigor from the project driven by the researcher, and information from the practical usage of VGT from the second project driven by industrial practitioners. Thus, this paper presents the results from two holistic case studies [69] from two different companies, in the continuation of this report referred to as Case 1, the study in Gothenburg, and Case 2, the study in Järfälla. The two case studies were conducted in parallel, but with different research units of analysis [69]. In Case 1 the unit of analysis was the VGT transition of one complete manual system test suite of a safety-critical air traffic management system, performed by experts from academia. In Case 2, the unit of analysis was instead the success or failure of the VGT transition project when performed in a practical context by industrial practitioners under real-world time and cost constraints. Consequently, Case
1 provided more detailed information about the VGT transition, whilst Case 2 provided information from a VGT transition project performed by industrial practitioners under practical conditions.

5.3.1 The industrial projects

Both VGT transition projects were conducted in industry with two mature industrial software products/systems. In Case 1, the VGT transition was performed on a distributed, safety-critical, air traffic management system with an excess of 1 million lines of code, highly configurable to satisfy customer needs, developed in a multiple programming languages. Additionally, the system was graphical user interface (GUI) driven with a very shallow GUI, meaning that most graphical bitmap components were continuously shown to the user, i.e. the GUI did not change much during interaction. The system has been developed using both plan-driven and iterative development processes, starting with requirements acquisition activities, followed by development activities and finally testing activities. For each activity a set of artifacts were developed, such as requirements specifications, design documents, user guides, test documentation, etc. However, for the study, the only document of interest was the manual system test descriptions, which specifies the manual test cases used for system testing. System testing that is generally performed once or twice every development iteration, i.e. once or twice every six months. Furthermore, the company’s developers are highly-educated and most have many years of industrial experience as different roles, e.g. as developers, testers and project managers. However, the company does not use dedicated testers, instead the developers perform the testing. The company has a hierarchal, yet flexible, organization, distributed between two locations, i.e. Gothenburg and Växjö. However, Case 1, due to resource constraints was only performed in Gothenburg. Furthermore, the system used in the project was developed according to a quality assurance process that is compliant with the RTCA DO-278 quality standard. A standard required by many of the system’s customers, which consist of both domestic, public and military, airports as well as international, public, airports.

In Case 2 the VGT transition was performed on manual test cases for a battlefield control system which is distributed over several computers and is both safety- and mission-critical. However, due to the limited involvement of the research team in Case 2, and confidentiality reasons, less can be disclosed about the system’s details. The product is however mature, i.e. it has been developed, maintained and deployed for many years. Additionally, the tested system is GUI-driven but with a deeper GUI meaning that the entire view of the GUI can change during interaction with the system, e.g. when opening menus or new windows. Development of the system was, at the time of the study, performed using an iterative process with regular manual testing performed by dedicated testers. However, in contrast to the system developed in Case 1, the tested system in Case 2 did not follow any quality assurance standard. The look and feel of the graphical bitmap components of the system’s GUI were however specified by a military standard. Finally, the product’s main customer is a Swedish military contractor.

Figure 5.1 visualizes the research process and the three parallel tracks that
Figure 5.1: Visualization of the research process showing three tracks. Track 1 contains activities performed in Case 1 and prior work at Saab AB in Gothenburg. Track 2 the activities of the research team and Track 3 the activities of Saab AB in Järfälla. Boxes that cross over the dotted lines were performed by the research team and the respective company.

were performed by the researchers in Case 1, practitioners in Case 2 and the support and data acquisition performed by the research team. The support activities were continuous in Case 1, with members of the research team on site, at the company, daily during the project. In Case 2, personal support was only given during two full-day workshops on site, whilst all other support activities were conducted over email or telephone. The workshops performed in Case 2 were also the main source of data acquisition from the company, e.g. information such as what type of system they were working with, what the Sikuli test architecture would look like, etc. In the second workshop, two semi-structured, one hour, interviews were held with the industrial practitioners to validate previously acquired information through triangulation. Consequently, the data acquisition in Case 2 was divided into three distinct phases, introduction (Workshop 1, in phase 1), data acquisition and implementation support through remote communication (Phase 2) and finally a retrospective analysis (Workshop 2, phase 3).

After the completion of Case 1 and Case 2, all collected/acquired information was analyzed and the CPLs identified. In Case 1, the analysis was done through a combination of discussions with the researcher who did the data collection and analysis of the thorough documentation that had been kept during the entire project. In Case 2, document analysis was used as well, but
the primary source of information came from the workshops and particularly the interviews since explicit questions were asked regarding the CPLs, and solutions to said CPLs. The two projects were performed mutually exclusive from one another, meaning that the leading researcher in Case 1 had no interaction with the industrial practitioners in Case 2. Thereby ensuring that any corroborating evidence from the two projects were not influenced by each other.

After identification, the CPLs were categorized into three tiers, with 29 mutually exclusive CPLs on the lowest level of abstraction, i.e. Tier 3. The Tier 3 CPLs were then generalized into 8 groups, Tier 2 CPLs, which could be grouped even further into three top tier CPLs, i.e. Tier 1 CPLs. Furthermore, the Tier 3 CPLs were analyzed to identify which were the most prominent CPLs in the projects. Prominence was evaluated based on occurrence in both projects, as well as more subjective measures, e.g. perceived negative impact on the transition, or usage, of VGT, added frustration to the VGT transition, and perceived external validity. In addition this analysis revealed four high-level generic solutions that had been identified during the two projects, which solve or mitigate roughly 50 percent of the identified CPLs. Cost, and return on investment, was also evaluated during the study based on the quantitative metrics that were collected from both cases and then analyzed. This analysis was performed in context of the acquired qualitative information, e.g. the industrial practitioner’s statements regarding VGT’s benefits and drawbacks, to rule out bias by ensuring that the qualitative and quantitative information was coherent. All information analysis was performed by another member of the research team than the one who performed the transition project in Case 1 to mitigate bias in the results. Finally, conclusions were drawn from the analyzed information, which were reviewed and validated by the researcher who did the data collection and industrial practitioners to eliminate bias introduced during the analysis.

5.3.2 Detailed data collection in Case 1

The VGT transition in Case 1 started with an analysis of the automated test scripts that were developed in our previous work at Saab in Gothenburg to identify what should/could be reused in the new project. In parallel with this analysis, a thorough analysis was performed of the manual test suite for the tested system. The document analysis was necessary because the version of the tested system differed from the system that had previously been used. In addition, the document analysis was required because the researcher who did the data collection was unfamiliar with the system and lacked the details of the previous work. A thorough documentation process was put in place at this stage of the project based around a set quantitative and qualitative metrics that were collected for each developed VGT script and/or the VGT test suite, including development time, execution times, CPLs, etc. Specifically, the information collection focused on sources and causes of CPLs that affected the VGT transition, e.g. was the CPL related to the script, the VGT tool or the system.

Even though the version of the tested system in Case 1 differed from the one used in our previous work, the core functionality of the new version was
the same as the old, i.e. airport landing and air traffic management. The main difference was that the new version only had one control position, whilst the old system had three, each with different capabilities for different operators. Consequently, the new version had limited functionality and analysis of the manual tests therefore showed that only 33 out of the 50 manual test cases, mentioned in previous work, were applicable. However, all of these 33, applicable, test cases could be automated and thereby constituted a full automated test suite, VGT suite, for that particular version of the system.

Furthermore, the system was distributed over three physical computers, which required the VGT tool, Sikuli, to be paired with a third party Virtual Network Connection (VNC) application in order to perform the test cases. Thus, the test system setup included three computers that were interconnected, through a local area network (LAN), to a fourth computer that ran a VNC viewer application and Sikuli. The test system setup has been visualized in Figure 5.2. Consequently, Sikuli, rather than to execute scripts locally, executed the scripts through the VNC viewer application to facilitate test script execution of distributed test cases that required interaction with more than one computer.

Once the analysis of the manual test suite, and the setup of the test system had been completed, an architecture was defined for the development of the VGT suite. The VGT suite architecture consisted of a main script that imported the individual test case scripts, and helper scripts, and executed these sequentially. The VGT scripts were implemented one at a time using the manual test cases as a specification, ensuring that each VGT script was a 1-to-1 mapped representation of the manual test case. This implementation choice was possible because the test steps in the manual test cases were defined in sequential scenarios. Hence, each test step of an automated test script became directly traceable to an equivalent manual test step in a manual test case. Additionally, to verify the automated test cases, the test scripts were executed after each test step had been developed, and verified against the results of corresponding manual test step. Furthermore, the automated test steps were defined in mutually exclusive methods, written in Sikuli script which is a scripting language based on Python. This approach made the scripts modular, allowing test steps to be reused. Modularity was one of the keywords for the transition projects to ensure reusability and maintainability of the scripts. Finally, after a test script had completed, and successfully executed against the tested system, it was integrated into the VGT suite. In situations where erroneous test script behavior was identified post-integration into the VGT suite, the test scripts were corrected and validated during execution of the entire VGT suite, i.e. execution of all the test cases in sequence. The validation was rigorously performed, even though time consuming, to ensure high quality, by comparing the outcome of each automated test step with the outcome of corresponding manual test step.

A second keyword for the VGT transition project in Case 1 was robustness. Robustness was achieved by implementing the scripts with three levels of failure redundancy for critical functions to mitigate catastrophic failure either due to image recognition failure, script failure, test system failure or detection of a defect in the tested system. In addition, all scripts were written as modular as possible to ensure reusability of generic functions. Modularization
also made it possible to run specific test steps out of order, thereby shortening script verification, since the entire test script did not have to be re-executed every time new functionality had been added or changed.

In addition, to increase the fault finding ability of the VGT suite, a third party screen-capture software was integrated into it. The screen-capture software was used when a script failed, which would cause the tested system to automatically reset to a known state after which the test scenario would be rerun whilst being recorded by the screen-capture software. The recording functionality simplified script verification and was also used to identify defects in the tested system, i.e. if a script failed, a new video clip was recorded and saved which the developers could view in order to recreate the defect. In addition to the he video-recordings, textual log files were created and saved for each executed test case.

### 5.3.3 Detailed data collection in Case 2

In contrast to Case 1, Case 2 was driven by industrial practitioners and started with a three week long evaluation of VGT as a technique. Three tools were evaluated during this period, i.e. Sikuli, eggPlant and Squish, to identify the most suitable tool to fulfill the company’s needs. After the evaluation, and because of recommendations from the research team, Sikuli was identified as the most suitable alternative for the automation.

Similarly to the transition in Case 1, the industrial practitioners in Case 2 used their manual test cases as specifications for the automated test cases. However, since the testers possessed expert domain knowledge, not every test case was implemented as a 1-to-1 mapping to the equivalent, manual, test case. The deviations from the 1-to-1 mappings were required since the manual tests in Case 2, defined as use cases, were not mutually exclusive, but rather linked together into test chains that contained several test flows, where each test flow constituted a test case. The manual test case architecture in Case 2 was however perceived to provide the VGT suite with a higher degree of flexibility and reusability than the manual test architecture used in Case 1.
However, similar to the VGT suite developed in Case 1, the developed VGT suite in Case 2 was based around a main script that imported individual test cases and executed these according to a predefined order that was specified by the user.

Furthermore, similar to Case 1, quantitative metrics and qualitative information was collected during the project. However, in contrast to the systematic, and rigorous, information collection in Case 1, it was performed in an ad hoc fashion in Case 2 because of time and cost constrains. The collected information was then conveyed to the research team, as shown in Figure 5.1, through e-mail, telephone or during the interviews in Phase 3.

### 5.3.4 The VGT suite

In this study we do not consider the developed VGT suites, or their architecture, part of the contributions of the work since the main focus of the study was on the challenges, problems and limitations (CPLs) that were identified during their development and usage. However, to provide background, and replicability, of the study, the following section will describe the developed VGT suites in more detail.

The VGT suites developed in Case 1 and Case 2 were similar in terms of architecture and were both built in Sikuli script, which is based on Python. Sikuli has support for writing individual VGT based unit tests and includes special assertion methods for unit testing. However, Sikuli does not have support for creating test suites of several individual unit tests. Hence, in order to create a VGT test suite of unit tests, using Sikuli’s supported functionality, all the tests have to be grouped into one large script, which has negative effects on reusability, maintainability, usability, etc, for large suites. Therefore, custom
test suite solutions were created in both projects by using Python’s support for object orientation and its ability to import scripts into other scripts. Therefore, both VGT suites consisted of a main script that imported the individual test cases and executed these according to an order specified manually in a list in the main script. The architecture for the VGT suites is visualized in Figure 5.3. As can be seen in the figure, each script was given a setup method, a test method, containing the test steps of each test case, and a teardown method. Consequently, the VGT scripts were developed to follow the same guidelines used for automated unit tests, e.g. JUnit [35]. Additionally, user defined methods, variables to setup the VGT suite, etc, were extracted from the individual scripts and put in a set of support scripts that were then imported to the main script and/or the test scripts that required them. The modular, and hierarchal, architecture helped shorten development time, increase reusability and improve maintainability of the scripts, since all reusable components were grouped in one location.

Test assertions were conducted through visual comparison between expected and actual output from the SUT using Sikuli’s image recognition algorithm. This was achieved using branch statements, i.e. if the expected output was observed the test step passed, else it failed.

The key difference between the VGT suite developed in Case 1 compared to Case 2 was how test result output was generated. In Case 1, the output was generated as textual log files using a custom solution that was spread across the main script and the individual test scripts. The solution documented the results of individual test steps but also summarized the results from the entire test case, i.e. providing feedback to the developers on two levels of abstraction. In addition, the output included video recordings of failed script executions, created using the third party recording software Camtasia.

In Case 2, the output was produced using an open source Python library that formatted the output from the test scripts into an HTML format, i.e. providing a graphical representation of the test results, similar to the output from automated unit tests, e.g. JUnit [35]. However, in contrast to Case 1, Case 2’s VGT suite did not record failed test scenarios, instead it only took screenshots of the tested system’s faulty state when a test case failed, i.e. capturing the GUI’s faulty state when a fault occurred.

The purpose of adding screenshots, and video recordings, to the VGT suites’ output was to provide the developers of the tested system with more information to simplify fault identification and recreation. This functionality also helped the testers to distinguish defects in the tested system from faults in the VGT suite itself, either caused by faulty test case implementation or image recognition failure during test script execution. Hence, mitigating the risk of false positives, i.e. reporting defects that were actually not defects.

### 5.4 Results and Analysis

During the study, 58 challenges, problems and limitations (CPLs) were identified related to the transition, or usage, of VGT. These CPLs were categorized post-project completion into three tiers of VGT related CPLs, as shown in Figure 5.4. Three main categories of CPLs were identified, i.e. CPLs related
to the tested system, the VGT tool or support software, constituting the Tier 1 CPLs. The Tier 1 CPLs were then split into eight CPL sub categories based on their origin and/or root of cause, i.e. the Tier 2 CPLs. These eight Tier 2 CPLs are: CPLs related to the version of the test system, the general test system, defects in the test system, CPLs related to the practices of the company, the tested system’s simulators, Sikuli, the test scripts or third party software. In Figure 5.4 these eight categories have been divided even further into a third level of abstraction, consisting of 29 identified mutually exclusive groups of CPLs, i.e. Tier 3 CPLs.

Out of the 58 identified CPLs, 14 were identified as Sikuli specific. Five of the 14 Sikuli CPLs were unique, i.e. mutually exclusive from any other Sikuli CPL, eight were related to image recognition failure or tool volatility and the last CPL related to the developed VGT test scripts. Furthermore, 20 out of the 58 identified CPLs were related to the version of the tested system, six related to the general system, i.e. the product, six were defects in the tested version of the system, one to the company in general and one to the simulator environment, i.e. 34 in total. Hence, more than twice as many of the identified CPLs originated from the tested system compared to the tool, i.e. Sikuli. The remaining 10 CPLs were related to the third party software that was used in order to realize the VGT suite, e.g. the virtual network connection (VNC) application used to implement distributed test cases and the recording software, Camtasia.

The detailed information regarding the CPLs were identified primarily in Case 1, but were corroborated by information acquired in Case 2. Hence, as can be seen in Figure 5.4, only 18 out of the 29 mutually exclusive CPLs were identified in both cases, i.e. 62 percent of the CPLs. However, the two cases were performed mutually exclusively of one another, meaning that the researcher who did the data collection in Case 1 had no contact with the industrial practitioners in Case 2. This study design intended to ensure that the collected information was based on the individual cases and could corroborate each other. Hence, provide evidence to support that the identified CPLs are generic for any VGT project performed with Sikuli.

In the following sections details regarding the identified CPLs will be presented. The presentation will use Case 1 as a base case but corroborating information from Case 2 will be presented as well.

5.4.1 Test system related CPLs

The identified test system related CPLs were split into five sub-categories, as shown in Figure 5.4. These CPLs are related to the version of the test system, the general test system, defects in the test system, specific to the company or the test system’s simulators. In this section, a summary of each CPL sub-category has been described, including data related to the individual CPLs of each category.

5.4.1.1 Test system version

Testing of complex systems with several versions and/or variants is related to several CPLs that also affect VGT. One such CPL relates to the manual test specification, e.g. that it can be faulty, out of date or developed for another
Figure 5.4: A hierarchal tree diagram over the Challenges, Problems and Limitations (CPLs) that were identified in Case 1 and Case 2. The CPLs have been divided into three tiers of abstraction, with Tier 3 being the lowest level where the 58 identified CPLs have been grouped into 29 groups of mutually exclusive groups of CPLs. The model continues to the right (The grayed out symbols), connecting to Figure 5.5, which shows potential solutions to the Tier 3 CPLs.

**SUT** - System under test, **VNC** - Virtual Network Connection, **Img. Rec.** - Image recognition, **GUI** - Graphical User Interface, **OS** - Operating system, **SW** - Software, **Func.** - Functionality.

version or variant of the tested system, i.e. not aligned with the tested system. This CPL appeared in Case 1 and was caused by the build, i.e. version, of system used in the project which was a demo system used to demonstrate the functionality of the product to potential customers, i.e. not a system intended
CHAPTER 5. PAPER D

for customer delivery. As a consequence, the demo system was limited in terms of functionality, lacked documentation, etc. The reason why this system was used in Case 1 was because of resource constraints. Whilst the product of the system required 12 computers, including three redundant servers, the demo system required only three computers in total. Furthermore, because the VGT suite was implemented as a 1-to-1 mapping of the manual test suite, not all of the system’s manual test cases could be automated. In our previous work [77] we performed automation with VGT on a version of the tested system that had 50 applicable manual test cases. However, for the product of the tested system, i.e. the full system, there are 67 manual system test cases in total of which only 33 could be automated in Case 1. Hence, restricting the usability and portability of the developed VGT test suite for other versions or variants of the product system. However, all of the 33 test cases were automated and thereby constituted a complete automated system test suite for demo system.

Lack of functionality that prohibited automation of more manual test cases were, but not limited to, the system version’s lack of operator roles, missing GUI components, missing radar functionality, missing simulator support, etc. This functionality had purposely been omitted by the company when the demo system was developed to scale down the amount of required hardware.

Furthermore, several CPLs were related to the manual test cases for the tested system, e.g. some test cases were incorrect or out of date. Other tests were found to be ambiguous or aimed at testing functionality that was no longer part of any variant or version of the system. These test cases were reported to the company and were considered a positive side effect of the automation since the faulty tests could be removed from the test specifications for several versions or variants of the system. Thus, this CPL relates to the complexity of keeping test specifications up to date as a mature system evolves into versions/variants over time.

Another identified CPL originates from the developers’/testers’ lack of domain knowledge in combination with ambiguity in the manual test descriptions/specifications. If a developer/tester performing the VGT transition lacks domain knowledge, it may be impossible for him/her to implement certain tests or he/she might implement them incorrectly due to ambiguities in the test specification, whilst a domain expert would have been able to resolve the CPL. This CPL was observed in Case 1, where the test specification that was used for the test automation was intended for another version of the tested system, making several of the test cases inapplicable. Hence, many of the test cases could only partially be implemented, or not be implemented at all, because the tested system lacked functionality. Furthermore, this made the VGT script implementation time consuming since the researcher who did the data collection continuously had to ask other developers at the company what the ambiguous test cases referred to, or what went wrong when the test cases did not align with the system specification. To resolve the CPL the test specification that was initially used in Case 1 was replaced with another version that was perceived to be better aligned with the system. However, it was soon found that the new version of the test specification included other test cases that could not be implemented on the version of the tested system used in the project, i.e. causing new CPLs. The new CPLs were not discovered before the automation had already started, once again due to the researcher who did
the data collection’s lack of domain knowledge. If the VGT transition had instead been performed by a domain expert, he or she might have been able to identify earlier that the test case was not applicable. Thus, saving time and cost. This assumption is supported by information from Case 2 where the industrial practitioner’s reported no such problems since they had extended knowledge of both the test cases and the SUT.

Several other CPLs were identified during analysis of the manual test specification, even though most of the CPLs were uncovered during script development. These CPLs were related to defects in the tested system that caused the VGT scripts to terminate with a certain probability or every time they were executed. During real-world execution of the VGT suite, i.e. during regression testing, a VGT script would terminate after identifying a defect, report the defect in the test output log, followed by a roll-back of the system to a known state before executing the next script. Consequently, a manual test case that identifies a defect at test step n can in practice only be automated up to step n since the defect would prohibit any interaction with the tested system after this step. This is because all further execution, after identifying a defect, would be within an unknown, and potentially useless, system state, making the interaction at test step n+1 invalid and/or useless compared to real-world use of the system. Furthermore, continued execution has a high probability of reporting false positive results because the test steps are generally dependent on one another, i.e. test step x sets the system in a state that is asserted in test step x+1 that also sets the system in a state that is required in test step x+2, etc. However, since the VGT suite, in Case 1, was developed for future use at Saab, all test steps had to be implemented regardless if they succeeded a faulty test step or not. Thereby, raising the usability and portability of the developed VGT suite for versions of the tested system where the fault had been corrected. To ensure that the test steps after a defect had been identified would still be performed during script execution, the assertion of the defect finding step, e.g. test step n, was disabled. However, since these test cases ignored the defects, the continued script execution, i.e. test steps n+1 and forward, potentially put the system in an invalid/unnatural state. Six defects were found during Case 1, by six different manual test cases, resulting in six partially implemented test scripts, i.e. scripts with disabled assertions. Hence, in future use of the test suite, for other versions or variants of the system, these assertions first need to be enabled.

The defect identification was continuous in four of the six developed test scripts that found defects. To ensure that these test scripts would execute all their test steps, the assertions that found the defects were disabled during verification of the scripts. This was possible because the test steps were mutually exclusive from one another and because of the shallow GUI design of the tested system that continuously showed all the GUI’s bitmap components to the user. However, in the remaining two test cases the tested system’s faulty behavior was not consistent. In these test cases the assertions were left active during verification of script functionality which gave the test scripts a certain probability of failure. This also made it possible to identify the faults in sufficient detail to report them to the company.

However, because of CPLs related to faulty system behavior, it is reason-
Table 5.1: Summary, and distribution, of problems, challenges and limitations (CPLs) related to the tested system.

<table>
<thead>
<tr>
<th>CPL category</th>
<th>CPL subcategory</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test System</td>
<td>Test System version</td>
<td>20</td>
<td>0.344</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (General)</td>
<td>6</td>
<td>0.103</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (Defects)</td>
<td>6</td>
<td>0.103</td>
</tr>
<tr>
<td>Test System</td>
<td>Test Company specific</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (Environment)</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>37</strong></td>
<td><strong>0.5844</strong></td>
</tr>
</tbody>
</table>

Table 5.1: Summary, and distribution, of problems, challenges and limitations (CPLs) related to the tested system.

It is also perceived that an incremental script development process in parallel with system development has benefits over a big bang script implementation. This assumption is based on the research into other automated test techniques and practices [15, 36], in particular Test-driven Development that suggests continuous, and incremental, test script development [101]. However, as shown in this study, a big bang implementation for legacy systems is also considered viable. We will return to the viability of this approach in Section 5.4.6, which discusses the return on investment of the VGT transition in Case 1 and Case 2. However, even though these assumptions are supported by research in other areas of automated testing, future work is still required to verify their affects on VGT.

5.4.1.2 Test system (General)

One of the most time consuming CPLs that was identified, in both Case 1 and Case 2, was that Sikuli, due to the speed of Sikuli’s image recognition algorithm, executes scripts faster than the system’s GUI can respond. Consequently, when Sikuli tries to perform an interaction, it is not certain that the GUI is ready to receive new input, which can cause the script to fail. This CPL is time-consuming because it requires the user to add delays in the scripts to synchronize the scripts’ execution with the tested system. Furthermore, this practice slows down the script execution time since the delays either have to wait for the GUI to reach a stable state or be based on fixed delays based on the worst case response time of the system. Furthermore, this practice, even though it adds robustness, usability, reusability and portability to the VGT suite it also adds maintenance costs since these delays generally have to be
maintained every time the system’s performance is changed. In addition, as reported by the practitioners in Case 2, this practice adds frustration to the VGT transition, especially for long test scenarios, since each synchronization, i.e. added delay, requires the script to be rerun to verify correctness. Consequently, if an added delay is not sufficient to synchronize the script with the system’s execution, the script once again has to be rerun. In addition, since the delays should preferably be as short as possible, but still ensure script robustness, this practice requires some trial and error which is both tedious and time consuming.

To mitigate this problem, most VGT tools, including Sikuli, have special methods that delay the script execution until a sought image is found, i.e. the script is delayed until a stable GUI state has been reached. However, these methods still require fine-tuning and therefore do not completely remove the synchronization CPL. For web-based systems, this CPL is especially problematic since network latency has to be taken into consideration when adding the delays, i.e. sudden dips in network latency, if not mitigated in the script, can cause the scripts to fail. Hence, a conclusion can be drawn that the execution speed of a VGT script is governed by the response time of the tested system. This conclusion is also corroborated by information collected in Case 1.

This problem became even more troublesome in Case 1 because it was observed that the tested system lost performance over time, i.e. its response rate deteriorated after a couple of days if the computers were not restarted. Similar reports were presented from Case 2 and had the effect that test scripts that had previously succeeded stopped working. However, given that the solution was to simply restart the computer(s) this CPL is considered minor. However, since only minor dips in response rate can cause the scripts to fail, which might be hard for a human to detect, this CPL is worth mentioning since it caused frustration during the project.

In addition, as reported by the practitioners in Case 2, a common comment by project managers and other developers viewing the automated script execution was: "Isn’t the execution faster than this?". This comment reflects, and reveals, an important perception about automated test techniques, i.e. the perception that all automated tests execute very fast. This perception is mostly true, e.g. automated unit tests can execute many hundreds, or even of thousands, of test case lines of code in minutes [36]. However, these automated test techniques, e.g. unit testing, approach testing from a lower level of system abstraction, e.g. through a white-box approach that stimulate the components directly, but out of context. In addition, these techniques do not provide interaction stimuli to the tested system in the same way as the end-user of the system. For instance, a unit test generally only tests one or a couple of components at once, thereby disregarding the timing constraints between the components that appear during actual usage of the system. Attempts have been made to use lower level test techniques, e.g. unit tests, for system test automation. However, these tests have quickly become very large and complex and therefore unmaintainable [14, 15]. VGT scripts take these timing constraints into consideration by emulating end user interaction, but, as mentioned, at the cost of test execution speed. However, based on information from both Case 1 and Case 2, VGT tests can still execute up to 16 times faster than their manual equivalent tests. Hence, VGT scripts execute considerably
slower than unit tests, which may be a CPL for industrial VGT adoption since the general perception of industrial practitioners, as mentioned, is that automated testing is very fast. However, what must also be considered is the higher level of system abstraction of these tests, which make them applicable for system and acceptance test automation.

Another CPL, which is general for all systems, relates to unexpected system behavior, which includes the behavior of the tested system itself, behavior of the operating system or third party software used to run the tested system or its surrounding environment. In Case 1, such CPLs appeared at several occasions but especially in test cases that required reconfiguration of the tested system. The reason was because reconfiguration required both the tested system and the computers’ operating system (OS) to be restarted. When the OS was shut down, one or several of the tested system’s services often crashed, causing the operating system to launch a pop-up message asking the user to terminate the process. The pop-ups blocked, or delayed, other components in the GUI from becoming visible and were thereby not available for the image recognition algorithm to find, causing the scripts to fail. Events, similar to this scenario can be handled using exception handling but it is often difficult, or even impossible, to anticipate when this type of unexpected behavior will occur, especially if the unexpected behavior is sparse. However, failure to mitigate these events can cause the entire VGT suite to fail, or report false positive faults. Scenarios when this type of behavior occur are when popups, e.g. software update messages or error messages are launched by the operating system, when system performance suddenly drops due to, for instance, a new process is started by the operating system, hardware failure or a defect in the tested system. To mitigate catastrophic script failure due to unexpected behavior, or because of identified defects, the VGT suite in Case 1 was implemented with triple failure mitigation redundancy. First, certain operations, e.g. delay operations based on image recognition, were encapsulated by exception handling blocks that, if triggered, tried to redo the operation on an individual test script level. Second, if the exception handling failed, the exception would be sent up one level in the VGT suite architecture to the main script that would try to roll back the system to a known state and then rerun the test script from the beginning. Third, if the rollback failed, the main script would restart the tested system to ensure a stable system state and then rerun the failed test case from the beginning. This solution added extra execution time to the overall test suite, but also added confidence that found defects were actually defects in the system rather than spontaneous image recognition failures or faults in the test scripts. Hence, if all three levels of failure mitigation failed to resume the test execution, the cause was in general ruled out to be a defect in the tested system. Similar practices were used in Case 2, but only with one level of failure mitigation.

5.4.1.3 Test system (Defects)

Many of the encountered defects, discussed in Section 5.4.1.1, were previously known to the company in Case 1. However, during the project, defects were also uncovered that were previously unknown or only partially known to the company. Hence, defects that had not been corrected in later versions or
variants of the tested system. The reason why these previously unknown defects were uncovered by VGT was because of the quicker and more cost effective execution of the VGT suite compared to manual tests. One reason was because some of the identified faults were not consistent, i.e. could not always be replicated, neither manually or automatically. For instance, one such defect regarded a tab menu in the tested system that did not always load properly. However, since there is no cost related to running the VGT suite it could be run several times in a row and thereby force the faulty system behavior to appear. In combination with the video recording functionality of the VGT suite in Case 1, the faulty behavior of the system could be determined and the defects could be reported in more detail to the company. Hence, showing how the video recording functionality of the VGT suite adds to its fault-finding ability and thereby usability. The faults that were uncovered were of different nature, from faulty GUI functionality, e.g. tabs not launching, to complex faults in the services of the tested system’s backend, e.g. missing alarms intended to warn the system user of incorrect or missing input from external interfaces. However, even though some of the defects could be considered minor, the tested system was safety-critical and therefore all faults or defects were considered critical.

Similar information regarding the identification of system defects was acquired from Case 2, where the speed and low execution cost of the VGT suite made it possible to identify three previously unknown defects in the tested system. However, in Case 2 the cause to why the defects had previously not been identified differed from the cause in Case 1. Previous to the automation in Case 2 the test cases were always executed in a linear order, i.e. starting with test case 1 to test case n, and only executed once every development iteration, which was generally six months to one year. Furthermore, due to strict testing budgets, it was not feasible to run all of the system’s manual test cases each development iteration. However, the automated tests, which were equivalent to these manual tests, could be executed several times every week and several times in row each time, which lead to the discovery of the previously unknown defects. Therefore, the practitioners in Case 2 reported that if it hadn’t been for VGT these defects had probably never been found.

Even though these examples show strengths with the technique, they also present a possible CPL, i.e. the CPL that a company may become so reliant on the technique that it is used as a substitute for manual testing. However, VGT is not perceived to be able to replace manual testing, primarily because the technique can only find faults that are specified in the test scenarios [15]. Hence, manual exploratory testing is still required in order to complement VGT to uncover new defects in the tested system [102]. These statements were also supported by the industrial practitioners in Case 2.

Furthermore, the information regarding defect and fault-finding indicates that the order of test case execution can affect the fault-finding ability of the scripts, e.g. executing test case b after a may may not have the same outcome as executing a after b. Thus, providing support for our previous statement, i.e. since there is no cost associated with running a VGT suite it can help uncover new defects by supporting execution several times in a row with different predetermined, or even random, test case orderings. Thus, perceivably, increase their effectiveness [8] by covering more system states and sequences.
of interaction that may appear during real-world usage of the system. However, some of these sequences may be invalid, i.e. never appearing in practical use, but potentially still add value to the development company by exposing what features of the system, in what scenarios, cause faulty system behavior. Consequently, the capability of VGT suites to be executed several times in different sequences, without additional cost, add to their usability. However, in order for the suites to do so effectively they require a good architecture. Thus presenting VGT suite developers with a CPL that is related to traditional software development rather than testing. A CPL that is common to most script-based test automation that also presents new CPLs, for instance how to design a test suite in a good and modular way and how to verify that the test code is correct. In particular, verification of test cases is a difficult CPL since the scripts need to be trustworthy, i.e. not find false positive and absolutely not false negatives. One way of performing verification of VGT tests is to verify them against the manual test descriptions, i.e. ensure that the VGT scripts perform in the same way as their manual test equivalents.

5.4.1.4 Test company specific CPLs

There was a mismatch between the tested system in Case 1 and the test specification that was used for the VGT transition. The researcher driving the automation in Case 1 also lacked the domain knowledge to identify this mismatch which later required several partial test cases to be refactored. However, the source of this CPL is not VGT specific but rather a general CPL that is related to the complexity of product management [103]. Product management that becomes increasingly more difficult as the number of artifacts, e.g. test specifications, increase in number for different versions and variants of a system, which can lead to confusion and faulty use, as experienced in Case 1. Hence, this CPL adds support to the conclusion that any VGT transition project should be driven by a domain and/or test system expert in order to mitigate that unnecessary effort is spent on development of VGT suites from faulty specifications.

In Case 2 another company specific CPL was identified that related to the VGT transition project’s limited budget. The limited budget forced the industrial practitioners to develop the first possible VGT script solution they could identify, which wasn’t necessarily the best in terms of script performance, reusability, etc. This CPL is common in industry and affects any process improvement and is therefore not VGT specific.

5.4.1.5 Test system (Environment)

Another CPL that was identified during Case 1 was related to the tested system’s simulators. The intended use of the tested system is to control the landing lights and radar equipment at an airport. However, since the actual hardware equipment, e.g. a radar station, is not available during development and testing at the company, the company instead uses a set of different simulators to stimulate the tested system’s external interfaces. Hence, simulating the hardware interfaces using software, which is a common practice in industry [17, 104]. These simulators are maintained by the development company.
5.4. RESULTS AND ANALYSIS

<table>
<thead>
<tr>
<th>CPL category</th>
<th>CPL sub-category</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Tool</td>
<td>Test tool (Sikuli)</td>
<td>13</td>
<td>0.224</td>
</tr>
<tr>
<td>Test Tool</td>
<td>Test scripts</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>14</td>
<td>0.2412</td>
</tr>
</tbody>
</table>

Table 5.2: Summary, and distribution, of problems, challenges and limitations (CPLs) related to the test tool (Sikuli).

and as the tested system has evolved so has the simulators, i.e. new, or additional, functionality has been added to the simulators to better represent the hardware interfaces connected to the system. Additionally, new simulators are constructed over time to test new functionality of the tested system that the previous simulators were not able to test, e.g. timing constraints on the input data which is sent over the external interfaces. However, not all simulators used at the company are compatible with all versions or variants of the tested system. Thus presenting a CPL regarding what test cases that could be automated for the version of the tested system in Case 1 since the test specifications specified the use of newer simulators than the system could support. Therefore, older versions of the simulators had to be used instead. Hence, restricting the usability, reusability and portability of the developed VGT suite as it is currently implemented. The costs required to refactor the VGT suite to work with newer simulators is also unknown and therefore a subject of future work. However, based on the research teams’ expert assessment, none of the interactions with the new simulators were determined to be unimplementable. Hence, there is no evidence to suggest that the migration to a new simulator environment would not be possible at the company. This expert conclusion is based on the fact that most of the company’s simulators mainly use standard Windows components that Sikuli, through empirical evaluation, has no problem to interact with.

5.4.2 Test tool related CPLs

The CPLs discussed so far have all been related to the tested system, or the development company, and are therefore perceived as CPLs that are plausible to appear in any VGT transition project, with any VGT tool. However, in both Case 1 and Case 2, the VGT tool Sikuli was used and the following section will present specific CPLs identified for that tool. Two sub-categories of CPLs were identified as Sikuli specific, i.e. CPLs related to the tool and CPLs related to the VGT suite that was developed.

5.4.2.1 Test tool (Sikuli) related CPLs

Sikuli is a tool developed and maintained in an open source project that was started at the User Interface Design Group at the Michigan Institute of Technology, but is now maintained and developed by the Sikuli Lab at the University of Colorado Boulder. The tool was at the time of the study still a release candidate, i.e. not a finished product. Consequently, the tool was volatile, con-
taining several defects that affected the stability of the tool’s image recognition algorithm, its behavior in general and its integrated development environment (IDE). The defects related to the tool’s IDE caused it to crash and freeze sporadically, forcing the user to restart the tool, which, as reported from the researcher who did the data collection in Case 1, and the industrial practitioners in Case 2, added frustration to working with the tool. However, as also reported by the practitioners in Case 2, there are ways of improving the tool’s stability, e.g. by running the scripts from the command-line rather than the tool’s IDE. No explicit reason was found why this practice raised the robustness of the scripts, but it was perceived, by the industrial practitioners, to be related to defects in the tool’s script engine.

During test script development it is often required to terminate the test script execution manually, e.g. due to recognized faulty behavior. However, manual termination of the script execution can cause Sikuli scripts to be corrupted. Hence, a serious CPL, which was identified both in Case 1 and Case 2, with impact on robustness and usability of Sikuli. In the instances where the CPL occurred, it was reported that the script logic could always be recovered but that the links to the sought images in the script all broke, which required recapturing of all the images. Hence, for larger scripts, this CPL can be both time consuming and tedious whilst also adding frustration. The mitigation practice found for this CPL during the project was once again to run the scripts from the command-line rather than from Sikuli’s IDE, which made it possible to terminate the script execution without any reported incidents. Thus, the recommendation, given by the industrial practitioners in Case 2, is to restrict the use of Sikuli’s IDE to script development only. Additionally, this CPL has been identified on both Windows and MacOs versions of the tool, but with perceivably higher frequency in Windows.

The most prominent Sikuli related CPLs are however connected to the tool’s image recognition algorithm, e.g. it randomly fails in cases where it has previously succeeded, randomly clicks on generic positions next to the sought images. These random failures were identified in both Case 1 and Case 2 to be the source of the most frustration during the VGT transition since the source of the problem has yet to be identified and appears completely random in nature. Image recognition failure often occurs during script development because the similarity level of the sought image is either too high or too low for the image recognition algorithm, which uses fuzzy recognition, to find a match, e.g. because there are several similar images on the screen. To ease the script development in these instances, Sikuli’s IDE has a built in feature that allows the user to preview what the image recognition algorithm considers a match to the sought image on the tested system’s GUI. This feature shows the developer where Sikuli has found a match on the screen, indicated with a square. If several matches are found, these are ranked according to the similarity of the found image compared to the sought image. The ranking is visually displayed with squares of different color, from light blue, for a match of low similarity, to bright red, for a match of high similarity. During script execution, Sikuli will always interact with the image with the highest similarity, if not told otherwise. According to the researcher who did the data collection in Case 1, and the practitioners in Case 2, corroborated by the authors’ own experiences, this feature has never failed to find a match, i.e. indicating a 100 percent
success rate of the image recognition algorithm. However, it was reported, from both cases, that even though the preview feature indicates that a match, of high similarity, is found in the tested system’s GUI, this does not ensure that the image recognition will find the sought image during script execution. Furthermore, even if the link to the sought image has been corrupted in the script, as described above, this feature still shows a match. The successful matching of corrupted images added a lot of confusion and frustration in both Case 1 and Case 2, since this feature’s behavior did not align with the behavior of the script, i.e. the feature indicated a match but the script failed.

Another identified CPL related to the image recognition algorithm was that it is often too lenient or too strict on what is considered a match to the sought image. Therefore, a lot of effort during script development is required to fine-tune, i.e. raise or lower, the similarity level of individual images, which in its default setting is set to 70 percent similarity. For larger scripts this practice becomes quite time consuming, especially since no pattern has been identified regarding which images require higher, or lower, similarity level than the default setting. Furthermore, as mentioned above, Sikuli’s scripts sometimes get corrupted, requiring images to be recaptured. Recaptured images do not retain the image similarity properties they had before they got corrupted. Hence, the similarity level once again needs to be set manually. Consequently adding further to the frustration to the script development. Furthermore, the need to fine-tune the similarity is considered to be one of the main contributors to development costs, but perceivably also maintenance costs.

Other CPLs related to the image recognition, identified in Case 1, concerns inconsistent ability to find certain bitmap objects, i.e. animated images and images with similar appearance but with slightly different color. The identification of animated images, e.g. blinking or moving images, was required since the tested system in Case 1 had animated buttons that indicated alarms or warnings to the user. For instance, in most airports there are nets that can be raised in order to stop an aircraft in an emergency, e.g. during brake failure. These nets are situated on opposite ends of the runway and should be raised dependent on from which direction the aircraft is approaching the runway. However, in case the wrong net is raised, or both nets are raised at the same time, the GUI warns the user by periodically switching the background color of the buttons, used to raise the nets, from red to yellow and back again in an interval of roughly one second. Testing this functionality, with confidence, showed to be problematic for Sikuli that often failed to find the buttons, depending on which state they were in. A conclusion can therefore be drawn that Sikuli is limited in its capability to interact with animated GUIs, i.e. lowering the tools usability. However, as mentioned, the tested systems, in both Case 1 and Case 2, were both quite static, i.e. mostly non-animated. Therefore, the CPL did not affect the overall VGT transition but rather just a few test cases.

In addition, the image recognition was, in some instances, unable to distinguish between images with similar but different color and similar appearance. For instance, the tested system in Case 1 is intended to be used both during the day and at night, i.e. during different lighting conditions. Hence, in order to make the GUI more comfortable to use during night, the GUI colors can be switched to a darker color set, effectively dimming down the brightness of the GUI. However, due to to limitations of Sikuli’s image recognition algorithm, it
was unable to test that that the GUI colors had been changed. The initial test was performed by asserting that the GUI with the lighter color set was not shown after the darker color set had been activated. However, even when the similarity level of the image of the lighter GUI was raised to 99 percent, the image recognition algorithm was still unable to differentiate the brighter from the darker appearance of the GUI. Consequently, the test case could only be partially implemented by identifying parts of the interface that changed more substantially, e.g. buttons that changed color from gray to yellow. However, this test still did not assure, with full confidence, that all aspects of the GUI were changed. Thus, this lack of confidence in the image recognition’s capabilities lowers the robustness, usability and portability of the scripts. This CPL may arise when, for instance, a test aims to verify that a button has changed its graphical state after being clicked, given that the images of the different states are similar.

Yet another CPL with Sikuli, experienced in both Case 1 and Case 2, concerns the lack of documentation about Sikuli’s scripting language. Sikuli’s scripting language, called Sikuli script, is based on Python but has been extended with a set of methods that make use of the image recognition capabilities of the tool. However, as reported by the researcher who did the data collection in Case 1 and the industrial practitioners in Case 2, there is no consistent, searchable, API for all of the methods supported by Sikuli script and what these methods’ properties are. As an example, Sikuli script has a method called wait, which originates in Python’s sleep function, which delays the script execution for t number of seconds. The method can take a series of different input parameters, i.e. wait(image, t), wait(t, image), wait(image) and wait(t). These input alternatives are not specified in the official Sikuli documentation but all have different behavior. The first alternative, wait(image, t), causes Sikuli to pause the execution for a maximum of t seconds or until it finds the sought image ”image”. However, if the second alternative, wait(t, image), is used, the script will always pause for t seconds and then try to find the image ”image”. Both functions are useful in different circumstances, but since the user generally takes the worst case scenario into account when setting the wait time, the second alternative will make the script execution time much longer. This was considered a large CPL in both Case 1 and Case 2 until it was discovered that the wait method had these different capabilities. In addition, this CPL, i.e. the lack of proper documentation for Sikuli script, can discard companies from using the tool. Especially since some of Sikuli script’s methods are unintuitive, e.g. the wait method. Hence, this CPL lowers the usability, learnability, reusability, portability and maintainability of the scripts. However, as we identified in our previous work [17], Sikuli script, or Python, is in general an intuitive programming language, even for novice users with limited programming experience.

Both the tested systems, in Case 1 and Case 2, included input areas that accepted swedish words as input. Hence, there were manual test cases that required the user to input swedish words and/or letters, i.e. å, ä, ö. However, Sikuli only supports an english keyboard for typing, i.e. the supported method “type” does not support swedish letters. Thus, in order to use the swedish letters, Sikuli’s “paste” method must be used instead, but this method does not work in all instances since it uses the operating systems (OS) clipboard
5.4. RESULTS AND ANALYSIS

which isn’t available, for instance when typing passwords into Windows OS login screen. The solution to this CPL, used by the researcher who did the data collection in Case 1, was to type ascii characters as combinations of pressing the ALT key followed by the ascii code of the letter. Identifying and implementing the solution was both time consuming and added frustration to the researcher. In addition, the fact that Sikuli does not support characters from other languages was puzzling since the tool’s IDE can be set to a variety of different languages, including swedish. Consequently, Sikuli scripts have some CPLs in terms of usability, and portability to systems with GUI’s in languages other than english.

Similarly, this CPL is present when the tool’s optical character recognition (OCR) algorithm is used. The OCR algorithm makes it possible to read texts of bitmap images but, once again, it only supports english letters. Hence, the swedish letter “å” is interpreted as an “a”, “ö” interpreted as an “o”, etc. Thus, once again limiting the usability and portability of the scripts.

Consequently, there are many serious CPLs related to the Sikuli tool, some of which are related to the tool’s IDE, others to the image recognition algorithm, etc. One cause of these CPLs has been determined to be because of the tool’s Java implementation, which, through exploratory experimentation on approximately 50 different computers, was found to be highly dependent on what version of the Java Runtime Environment (JRE) is installed. In addition, the current version of Sikuli, at the time of writing this report, requires Java 6, i.e. JRE 6, and is only stable for certain versions of said JRE. However, no comprehensive evaluation has been performed to find which versions of JRE 6 that make Sikuli more stable, but is a subject of future work. In addition, for users of Java 7, Sikuli’s initialization file has to be modified to use the exact path to the JRE 6 executable, rather than the path provided by the operating system’s environmental variables. Thus, many users, especially users with limited programming and/or OS knowledge, can be discouraged from using the tool, if they get it to work at all. In addition, our exploratory experiments could also show that Sikuli is more stable, in general, on MacOs than on Windows. Consequently, this CPL limits the robustness, usability, portability and maintainability of the scripts and the tool.

5.4.2.2 Test application

The VGT suites that were developed in Case 1 and Case 2 were both developed as 1-to-1 mappings of the manual test cases. However, the manual tests were, in both cases, designed to be as unambiguous and simple as possible for a human tester to perform. Hence, the test cases were defined in mutually exclusive steps that each consisted of setup of the system, through manual input, to put the system in a specific state before an assertion, i.e. the test, was performed, followed by another setup, input, etc. Thus, the tester would start with test step 1 in test case x and then, without requiring any knowledge of future steps, execute each step of the test case to test step n. The benefit of this approach is that any tester, or even developer, can perform these tests. The drawback is that this approach may be time consuming, since the tester has to jump back and forth between, for instance, a simulator and the tested system several times, i.e. set up the simulator, do the test, do a new setup,
etc. However, the test steps, in Case 1, were mutually exclusive, meaning that all the setup of the simulator could have been done in one step rather than several to save time. Grouping all of the simulator setup steps had however made the test case more complex and could potentially have added ambiguity to the test case. However, when executing a test case automatically, ambiguity in the script becomes less of a concern due to the more structured semantics of a script compared to natural language. Furthermore, said complexity is not a problem for the computer, only the human interpreter. Thus, there are other aspects than ambiguity to consider when performing automated testing, which are more important such as script performance, quality and reusability. Consequently, a 1-to-1 mapping between manual tests and VGT scripts is not necessarily the best approach. On the one hand, the 1-to-1 mapping approach allows the scripts to be verified through comparison with manual test execution, but on the other, it can have negative effects on the performance of the script. An alternative automation approach is therefore to group all related interactions with similar GUI component in one test step, given that they are mutually exclusive and do not affect the flow of the test scenario. The benefit is that the execution time becomes lower, whilst still allowing the developer to verify the test script outcome with the manual test case. The CPL lies in identifying these mutually exclusive test steps and group them together, correctly, in the script. Furthermore, this practice contains a trade-off since it raises the maintenance costs of the scripts because changes to the manual test cases become harder to update in the scripts. Another potential automation approach is to disregard the manual tests all together. Hence, instead of using the manual tests as a specification use domain expertise to build an automated test suite for the core functionality of the system. The drawback of this approach is that it requires domain and system experts to write the scripts, which might be a CPL due to the associated cost. An alternate, inverted, approach is to only automate the test suite’s large and complex test cases, e.g. test cases that are prone to faulty execution by a human, or test cases that are so long that they become cumbersome for a human to execute. These alternative automation approaches are supported by information acquired from the practitioners in Case 2 that in the future will focus on developing a more generic VGT application to test all the basic functionality of the tested system, i.e. not following the manual test specification.

Another, more concrete, CPL, related to the VGT suite, which was identified in Case 1, consisted of a combination of how Sikuli uses the mouse cursor and the speed of the tool. In order to mark and copy a generic text from an application, a human can double-click one the text and then copy it using a keyboard shortcut. This functionality was required in some of the test cases in Case 1. However, since Sikuli performs the double-click with such high speed, i.e. much quicker than a human, the operating system did not always register both of the clicks, which caused the script to fail. This CPL is minor, and can be solved by changing Sikuli’s settings to lengthen the time between clicks. However, the CPL is still worth mentioning because even though Sikuli, as all other VGT tools, interacts with the tested system in the same way as a human, it is not human. Thus, the developer needs to consider what Sikuli is actually doing, underneath the hood, when the script is being developed and executed to avoid CPLs that originate from Sikuli acting “non-human”.

5.4. RESULTS AND ANALYSIS

<table>
<thead>
<tr>
<th>CPL category</th>
<th>CPL sub-category</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support software</td>
<td>Third party software</td>
<td>10</td>
<td>0.172</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>Total</strong></td>
<td><strong>10</strong></td>
<td><strong>0.172</strong></td>
</tr>
</tbody>
</table>

Table 5.3: Summary, and distribution, of problems, challenges and limitations (CPLs) related to the support software, e.g. VNC.

An alternative solution to the text marking CPL, discussed above, which was found to be more robust, is to search for text using different applications search functions, e.g. in text editors. The search function was used in Case 1 in test cases that required XML files to be rewritten in order to change the layout of the tested system’s GUI, since a found match to a search, generally, automatically marks the found text. The drawback of this approach is that you have to know what text you are looking for, and it only works in systems that have a search function. A third alternative solution is to use Sikuli’s Optical Character Recognition (OCR) algorithm that allows the tool to transform text in images to strings that can be used for further processing in the script. However, Sikuli’s OCR algorithm was found to be unreliable in the available version of the tool, at the time the study was performed, and was therefore used as sparsely as possible.

5.4.3 Support software related CPLs

As shown in Figure 5.2 the test system in Case 1, included several computers connected through LAN, which Sikuli interacted with using a third party virtual network connection (VNC) application. A similar setup was used in Case 2, but only between two computers rather than four. The reason for using VNC was two-fold. First to allow Sikuli to perform distributed test cases, i.e. test cases that required interaction on different computers. Second to make the Sikuli’s script execution non-intrusive, i.e. removing the impact of running the performance intensive image recognition on the same computer as the tested system. Non-intrusiveness helps mitigate the CPL that the image recognition algorithm might steal computational resources from the tested system, which could potentially change the tested system’s behavior during runtime, i.e. slow down the execution. Thus, put the system in states that will not occur during real-world use or cause the test scripts to fail due to failed synchronization between the scripts the tested system.

However, the use of VNC was also the cause of several CPLs. First, when Sikuli is executed locally it can remove the mouse-pointer from the screen, making the mouse-pointers location irrelevant for the success of an image recognition. However, when Sikuli is executed over the VNC application, the mouse pointer cannot be removed since it is rendered remotely on the target computer. The mouse pointer can therefore obstruct buttons, and other sought bitmaps, thereby causing the image recognition to fail. Furthermore, the use of VNC had degenerative effects on the success rate of the image recognition because of network latency. Even when the VNC application was running in
an optimized setup, the frame rate of the viewer caused the image recognition to fail. This CPL was especially troublesome for test cases that included interaction with animated graphical components, e.g., the emergency nets in the tested system in Case 1, since the lower frame-rate could cause the buttons to be distorted as they were toggling color. Hence, on occasion, the buttons were rendered with the top half of the button in red color, and the button half in yellow color, or vice versa. In addition, VNC introduced other, less obvious, faulty behavior. The increased faulty behavior was observed in both Case 1 and Case 2, but VNC was identified as the source of the CPLs in Case 2. Hence, even though running Sikuli over VNC increases the tool’s usability, it also lowers the tool’s robustness since this practice increases the chance of image recognition failure.

In addition, as found in Case 1, the choice of VNC application is a relevant factor. At the start of the project, a more simplistic VNC application was used, but it was soon discarded due to poor performance and because it was unable to send keyboard commands to the tested system. These keyboard commands, e.g., CTRL+ALT+DELETE and CTRL+V, where used to simplify, and/or, where required, to perform some of the test cases. Another VNC application was therefore acquired, which solved the CPL regarding the keyboard commands and also increased the stability and speed of the remote image transfer. Consequently, even though VGT tools can interact with any third party software, the developer should, if there are multiple software options, seek to find the one most compatible with VGT.

Furthermore, the VNC application sometimes lost its connection, which made the screen freeze or caused the application to minimize, which caused the scripts to fail as well. This CPL was experienced in both Case 1 and 2, but no solution was found to resolve it. In addition, as also identified in both cases, the VNC application sometimes distorted the colors of the SUT’s GUI, typically during start-up. This CPL was solved in Case 1 by adding script functionality that restarted the VNC application when the distortion appeared.

In Case 1 a third party recording software, Camtasia, was, as mentioned, added to the VGT suite application. The tool was used to capture recordings of the test suite execution which can help developers to identify the cause of faults in the system and recreate the faults. However, during the project, in several occasions the software could not be started during script execution, which resulted in the VGT suite terminating before all the test cases had been executed. This CPL shows that even though VGT is able to interact with any bitmap component on the screen, precautions still have to be taken that said interaction is robust, in all aspects of the developed VGT suite.

5.4.4 CPL Summary

58 challenges, problems and limitations (CPLs) where identified during this project, primarily through analysis of the information collected in Case 1, at Saab in Gothenburg, corroborated by information provided from Case 2, i.e. Saab in Järfaåla. In the analysis the CPLs were categorized into three tiers, with the lowest, Tier 3, containing 29 mutually exclusive groups of CPLs, as shown in Figure 5.4. Table 5.4 summarizes, numerically, how these 29 groups
Table 5.4: Summary of distribution of problems, challenges and limitations (CPLs) that were identified during the automation process ordered according to occurrence of the Tier 2 CPLs. Tier 1 CPL-categories have been listed for each sub-category.

were divided over the Tier 2 CPL categories, and in turn how the Tier 2 CPLs were divided over the top three Tier 1 CPLs. The top three CPLs concern either the tested system itself, the test tool or support software not directly connected to the tested system itself, e.g. the third party virtual network control (VNC) software. Analysis of the collected CPLs shows that most of them relate to the tested system itself, rather than the testing tool, i.e. Sikuli. However, the most prominent CPLs were determined to concern the tool and its image recognition algorithm, which sometimes failed unexpectedly, had limited ability to interact with animated graphical GUI bitmaps, etc. In Table 5.5, the eight most prominent CPLs have been summarized together with their impact on VGT’s applicability in industry or the quality of a VGT suite. These eight CPLs were chosen based on occurrence during the project, but also more subjective measures such as added frustration, confusion, etc.

5.4.5 Potential CPL solutions

The focus of this work is on the CPLs related to VGT when performed in industrial practice but four generic solutions were also identified. These solutions have been summarized in Table 5.6 together with the Tier 3 CPLs, from Table 5.5, that they solve or mitigate. The reason for the low number of presented solutions, in this report, is because many of the solutions that were found/used in the study were ad hoc and thereby not generalizable. In Figure 5.5 the four generic solutions have been mapped to the Tier 3 CPLs. As can also be seen from Figure 5.5, not all of the CPLs are listed, which is either because no generic solution was identified to solve them, or because no solution was found at all. However, as shown in Table 5.6, and Figure 5.5 these generic solutions are applicable for solving or mitigation of more than 50 percent of the CPLs.
<table>
<thead>
<tr>
<th>Nr</th>
<th>Title</th>
<th>Description</th>
<th>Q-attr.</th>
<th>ID. at</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Image recognition volatility</td>
<td>The image recognition algorithm randomly fails for reasons unknown causing failures in scripts that previously worked. This problem is assumed to be related to Sikuli not being a finished product.</td>
<td>Rob, Usa.</td>
<td>Case 1 and Case 2</td>
</tr>
<tr>
<td>2</td>
<td>Image recognition limitations</td>
<td>The image recognition algorithm has limited ability in finding certain animated objects and differentiating between objects with similar color, e.g. dark to light gray.</td>
<td>Rob, Usa.</td>
<td>Case 1</td>
</tr>
<tr>
<td>3</td>
<td>Negative VNC effects</td>
<td>Running Sikuli through a VNC application allows testing of distributed systems, however it also increases the chance of image recognition failure due to VNC updates, color changes in the GUI, mouse cursor placement, etc.</td>
<td>Rob, Usa, Port.</td>
<td>Case 1 and Case 2</td>
</tr>
<tr>
<td>4</td>
<td>Sikuli IDE volatility</td>
<td>Sikuli is not a finished product and therefore has faults that cause it to crash, loose links to images in scripts, fail to start, etc. Note that these CPLs do not include image recognition related CPLs.</td>
<td>Rob, Usa, Port, Reu, Main, Mod.</td>
<td>Case 1 and Case 2</td>
</tr>
<tr>
<td>5</td>
<td>1-to-1 mapping</td>
<td>Using manual test cases as the specification for the automated scripts is not always feasible nor appropriate in terms of performance, maintainability, etc. Furthermore this documentation can be faulty.</td>
<td>Usa, Port, Reu, Main.</td>
<td>Case 1 and Case 2</td>
</tr>
<tr>
<td>6</td>
<td>Test system limitations</td>
<td>Test system CPLs appear on a case by case basis and are hard to generalize. A reoccurring CPL is however that the tested system is usually slower than the script execution, e.g. requiring delay statements to synchronize the scripts and the tested system. This is especially true for web systems.</td>
<td>Rob, Usa, Reu, Port.</td>
<td>Case 1 and Case 2</td>
</tr>
<tr>
<td>7</td>
<td>Hardware limitations</td>
<td>Image recognition is a performance heavy operation and lack of hardware support therefore influences the success rate of the image recognition, especially in real-time systems with animated interfaces.</td>
<td>Rob, Usa, Reu, Port.</td>
<td>Case 1</td>
</tr>
<tr>
<td>8</td>
<td>Test system maturity</td>
<td>The scripts require the images of the interaction components of the tested system’s GUI. These can be captured from the tested system directly given that all functionality of the test has already been implemented. Faulty behavior of components will be found during automation but also hinders completion of the script scenarios.</td>
<td>Rob, Usa, Reu, Port.</td>
<td>Case 1 and Case 2</td>
</tr>
</tbody>
</table>

Table 5.5: A summary of the eight most prominent challenges, problems and limitations (CPLs) encountered during the projects, i.e. Case 1 and Case 2. These CPLs were chosen based on their prominence during the projects, measured based on occurrence, perceived negative impact on the transition to, or usage of, VGT, added frustration, etc. Rob - Robustness, Usa - Usability, Reu - Reusability, Learn - Learnability, Port - Portability, Main - Maintainability.
5.4. RESULTS AND ANALYSIS

The first generic solution, used in both projects, was to ensure that the scripts were developed with redundant levels of exception handling to mitigate CPLs such as tool and image recognition volatility, etc. This exception handling was achieved by using Python’s inherent exception handling. Developing this exception handling is however quite time consuming and can be complex since these exceptions are most often caused by unexpected tool or system behavior. However, the solution, which was used in both Case 1 and Case 2, was reported as effective.

The failure mitigation in Case 1 consisted of three levels of failure redundancy, i.e. on a method level, on a script level and on a test suite level, as previously described in Section 5.4.1.2. Hence, if the script would fail, the failed interaction method would first be rerun, which if failed again would result in the entire test case be rerun, and finally if that failed as well, the test system would be restarted and the script rerun a third time. For each rerun a textual log of the test execution was automatically produced and for every rerun above the method level, the execution was also recorded. The failure mitigation in Case 2 was less complex, with only one level of redundancy. Hence, if a test case failed, the system would roll back the system, and then continue with the next test case. Rollbacks were in both cases performed with a teardown method similar to the JUnit test framework [35]. This solution is perceived as generic given that the VGT tool which is used for the automation has similar scripting support and solves CPLs such as script failure due to unexpected behavior of the tested system, the operating system or supporting software, or image recognition failure, etc. Some VGT tools also have other types of redundant failure mitigation, for instance several image recognition algorithms and image repair features.

The second solution that was identified, which mitigates the lack of Sikuli documentation, is to continuously document the script development, e.g. document the test suite architecture, the functionality of help scripts and methods. Thus, ensuring that new testers, and/or developers, can more easily start working with the test suite, but more importantly, to mitigate degradation of the VGT suite over time [15]. This solution shows how VGT testing has commonalities with traditional software development. However, documentation is only explicitly required for the test architecture, since the scenario-based scripts are generally intuitive by themselves. This intuitiveness comes from the combination of high level of interaction, which is equal to human user usage of the system, and images in the scripts that define with what these interactions are performed. In addition, given that the scripts are implemented as 1-to-1 representations of the manual test cases, the manual test case descriptions also serve as specifications, and documentation, for the test scripts.

The third identified solution regards the removal, or non-usage, of remote computer control software, e.g. virtual network connection (VNC) applications. Removal of VNC from the test architecture raises stability of the test execution by mitigating detrimental effects to the image recognition due to network latency, lowered frame-rate, etc. However, this practice is a double-edged sword, because, even though it raises robustness, it also restricts the number of test cases that can be performed on distributed systems. In addition, by running the test scripts locally, more load is put on the computer running the tested system. Hence, raising the risk of faulty behavior of the tested system.
due to the lack of performance resources, e.g. access to the central processing unit (CPU) or the computers random access memory (RAM). However, as reported by the practitioners in Case 2, by running the test suite locally, a success-rate of close to 100 percent could be achieved, whilst with VNC, in their context, only a success-rate of 70 percent could be achieved. Consequently, the use of VNC allows VGT tools, e.g. Sikuli, to automate manual test cases for distributed systems, but potentially also lowers the success-rate of the image recognition algorithm.

The fourth solution aims to solve the CPL that VGT scripts generally execute quicker than the tested system can update its GUI, i.e. the scripts are not properly synchronized with the tested system. This CPL was reported, in both Case 1 and Case 2, to be a huge source of frustration during the automation and also very time consuming since no explicit pattern could be identified when and where delays had to be added in the scripts to synchronize them with the tested system. However, as reported by the researcher who did the data collection and industrial practitioners, this CPL could be mitigated through systematic insertion of delays in the scripts at locations which could, later during the project, be estimated upfront based on experience of working with the tool. In Case 1, this solution was also supported by the development of custom methods with an additional time delay parameter. For instance, the click(img) method from Sikuli’s instruction set, where “img” would be the sought image, was expanded to create a click(img, delay) method which delayed the script execution “delay” number of seconds before performing the click. These custom methods provided additional robustness to the scripts but also increased execution time since these methods required the sought image to be found twice, first by Sikuli’s wait for image function and second by the click function, i.e. doubling the minimum number of required image recognition sweeps. However, due to the increased robustness, the researcher who did the data collection in Case 1 reported that it was still beneficial, especially since the image recognition algorithm in Sikuli is quite fast, i.e. can perform upwards of 5 complete image recognition sweeps of the computer monitor per second. This solution is proposed as generic since most VGT tools provide methods that can wait for the system to reach a stable state before the execution continues.

Finally, even though several generic solutions and mitigation practices were found for the CPLs, there were still CPLs that required ad hoc solutions. These solutions were specific to the two projects and could therefore not be generalized. Some of these solutions have been mentioned in Section 5.4. In addition, there were some CPLs that could not be solved or mitigated because they required larger effort to be solved and where therefore out of scope for this study, e.g. changes to the development company’s documentation process, and were therefore out of scope for this project. The unsolvable CPLs can be grouped into two categories. First, CPLs for which potential solutions could be identified, but which required so much effort that they were out of scope for the project. Second, CPLs where no solution was identified at all, e.g. how to ensure alignment between the test specification and the tested system.
### Table 5.6: A summary of general solutions that were identified for the CPLs listed in Table 5.5.

<table>
<thead>
<tr>
<th>Nr</th>
<th>Title</th>
<th>Description</th>
<th>Solves problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Redundant script failure mitigation</td>
<td>Adding failure mitigation code in the scripts, i.e. exception handling, on several levels of the script can help mitigate tool and image recognition volatility as well as unexpected system behavior.</td>
<td>Image recognition volatility, Image recognition limitations, Negative VNC effects, test system limitations.</td>
</tr>
<tr>
<td>2</td>
<td>VGT script documentation and user defined methods</td>
<td>In order to raise script robustness, usability, reusability, etc., it is recommended to construct user defined methods and other reusable artifacts. However, in order to preserve the knowledge of these artifacts, they should be thoroughly documented to make it easier for new developers to start working with Sikuli script.</td>
<td>Sikuli IDE volatility, 1-to-1 mapping.</td>
</tr>
<tr>
<td>3</td>
<td>Local/Remote script execution</td>
<td>Removing the remote system control application, e.g. virtual network control (VNC) application, from the VGT suite makes it possible to raise the stability of the script execution. However, this also limits what test cases can be executed for distributed systems.</td>
<td>Negative VNC effects, 1-to-1 mapping.</td>
</tr>
<tr>
<td>4</td>
<td>Systematic SUT synchronization</td>
<td>The VGT suite often executes quicker than the tested system reacts to input and therefore requires the scripts to be delayed. By creating custom methods that take timing input and smart usage of the ”wait for image method” in Sikuli script, a process can be constructed to iteratively synchronize the script with the tested system.</td>
<td>1-to-1 mapping, test system limitations.</td>
</tr>
<tr>
<td>5</td>
<td>Other or no solution</td>
<td>For several of the CPLs ad hoc solutions had to be formulated. These solutions included, but were not limited to, analysis and replacement of the manual test suites, hardware and software reboots to reset the tested system and/or Sikuli, change of VNC application, development of partially implemented test cases, etc. For the sake of completeness, we also state here that there were CPLs that could not be solved, e.g. the missing system functionality, stopping Sikuli from getting corrupted, etc. Hence, there are CPLs that should be investigated in future work to be solved and/or mitigated.</td>
<td>Image recognition volatility, Image recognition limitations, Negative VNC effects, Sikuli IDE volatility, test system limitations, Hardware limitations and test system maturity.</td>
</tr>
</tbody>
</table>
5.4.6 Defect finding ability, development cost and return on investment (ROI)

Thus far, this report has focused on the CPLs related to the transition and usage of VGT for high system level test automation, and the number of CPLs have been considerate. However, the industrial practitioners in Case 2, still stated that VGT, performed with Sikuli, is a valuable and cost effective technique. They reported that they did not encounter any functionality that they could not automate using Sikuli, only that it was more or less challenging. However, since VGT is a test technique, the primary measure of successful application is still the defect finding ability of the developed VGT suites. For the tested system in Case 1, six different defects were identified.

[A] Switching between military and civil landing lights did not work.
[B] The button to switch military and civil landing lights did not disable as intended.
[C] The button to switch military and civil landing lights did not disappear as intended.
[D] Switching quickly between runways caused the tested system to freeze.
5.4. RESULTS AND ANALYSIS

[E] Tabs for switching between views would not load.

[F] Logging out of Windows caused the tested system to freeze (The main thread of the tested system would not terminate).

These defects were reported in detail to the company and have since the study been corrected and are no longer present in any commercial version or variant of the system.

These defects were identified during implementation or execution of the VGT suite. Analysis of the found defects found that four of the defects were previously known to the company and already corrected in later versions of the system, whilst two were still unknown. Further analysis showed that the manual test cases could identify all six defects, but since two of the defects were sporadic it required several tries to force their faulty behavior and therefore they had not previously been found. Hence, the VGT scripts were able to identify all the defects that the manual tests cases applicable for the tested system could identify, i.e. providing the VGT suite with the same level of confidence, in terms of defect finding ability, as the manual tests.

In Case 2, the industrial practitioners reported similar results, i.e. that their automated tests could identify all the defects that the manual tests could identify. Additionally, as mentioned, they reported that the VGT suite could identify three defects that previously had been unknown to the company. In the past, the manual test cases had only been run once in sequence every development iteration, i.e. starting with test case 1 and ending with test case n. Furthermore, due to budget constraints, not all of the system’s manual test cases could be applied each iteration. However, by automating the system tests using VGT, it became possible to run the test cases more often and therefore not only provide higher quicker feedback to the developers, but also cover more test cases. In addition, the automation made it possible to run the test cases in several different orders, which resulted in the VGT suite finding the previously unknown faults. Hence, these results show the importance of what order the individual test cases are executed. Furthermore, since the execution order of the automated test cases is simple to change, VGT allows the user to quickly and cost-effectively cover more meta-level scenarios, at almost no additional cost. However, for this practice to work, the test cases need to be independent, such that they can be run out of order. In addition, it is important that the teardown, or rollback, methods of each script are well defined to put the system back into known states before the next test case is executed, to mitigate any side-effects of failed tests [35]. Failure to rollback the system can result in testing of invalid states and thereby reporting defects that are false negatives.

Table 5.7 summarizes some of the cost metrics that were acquired in Case 1 and Case 2. Worth mentioning, again, is that Case 1 was performed with only one researcher whilst Case 2 was performed by two industrial practitioners. In addition, the researcher in Case 1 had limited development and testing knowledge for the test system whilst the practitioners in Case 2 were domain experts. Furthermore, only one manual test suite was automated in Case 1, whilst in Case 2 a total of three suites were automated. Two of the test suites in Case 2 were considered to be more complex than the average test suites used at the company, whilst the third was considered equal in complexity compared
to the average. Additionally, the manual test suite that was automated in Case 1 was perceived to be of roughly equal complexity as the more complex test suites in Case 2. Here, complexity was evaluated based on the number of test steps of each test case, importance of test case success, complexity of test cases GUI interactions, etc.

Whilst the test suite in Case 1 was built around tables that defined test scenarios with defined input and expected output for each test step, defined on each row of the table, the test cases in Case 2 were built around more loosely coupled use cases. These use cases, examples shown in the right of Figure 5.6, were then tied together on a meta-level to form test scenarios, as shown in the left of Figure 5.6. Figure 5.6, on the left, exemplifies a test chain used in Case 2 which is made up from three different test flows, or scenarios. These scenarios all start with use case 1 and 2, i.e. UC 1 and UC 2, and are then followed by one out of three exchangeable use cases, i.e. UC 3A, 3B or 3C (Middle of Figure 5.6). However, the manual test structure also allowed test scenarios to be of unequal length, exemplified with UC 3AA, bottom left of Figure 5.6. The three different test chains are then joined again, and completed, by UC 4 (Bottom of Figure 5.6). This structure perceivably improved the manual test cases usability, maintainability and reusability since the use cases could simply be switched out in any part of the chain to test newly added functionality of the system. The drawback of this approach, for manual testing, is that many of the test scenarios become very similar and therefore tedious to test, i.e. the tester has to perform the same interactions over and over. However, once automated, the tediousness no longer becomes a problem but the benefits, e.g. reusability and maintainability, are kept intact, since new scripts can easily be formed by reusing the individual use cases together with a newly developed scripts.

Table 5.7 shows the development time of the VGT suites in Case 1 and Case 2. As can be seen from the table, the development time in Case 1 was considerably lower than the development time in Case 2. However, the time in Case 1 is based on very precise measurements, performed by the researcher.
5.4. RESULTS AND ANALYSIS

<table>
<thead>
<tr>
<th>Project</th>
<th>Dev. time (mh)</th>
<th>Nr. of Man. test suites</th>
<th>Dev. time per suite (mh)</th>
<th>Maintenance (mh)</th>
<th>Manual suite Exe. time (mh)</th>
<th>VGT suite Exe. time (h)</th>
<th>Pos. ROI after executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>213</td>
<td>1</td>
<td>213</td>
<td>-</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>Case 2</td>
<td>1032</td>
<td>3</td>
<td>344</td>
<td>266</td>
<td>80</td>
<td>2.5</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.7: Summary of quantitative metrics collected during the VGT transition projects in Case 1 and Case 2. **mh** - man-hour, **h** - hour

...doing data collection, who reported of the actual time spent on script development in rigorous detail. In contrast, in Case 2 the development time was measured by industrial practitioners in a real-world context where measuring the exact development time was low priority compared to the development itself. Consequently, the measured time from Case 2 contains more overhead, i.e. time not spent on development, than the measured time in Case 1. Hence, the number of implemented test suites, the manual test architectures, expertise of the script developers, and time measurement methods, all differed between the two cases. Therefore, the data in Table 5.7 concerning development time are from two different contexts and should therefore be treated as such, i.e. not compared directly without taking the context, and measurement methods, into account.

However, Table 5.7 does show some interesting, and comparable, differences in terms of improved execution time between the manual suites and the developed VGT suites. In Case 2, the improved execution time was quite considerable, i.e. by a factor of 16, whilst in Case 1 the improvement was only marginal, i.e. 1.06. The reason for this difference can be found in the individual test cases, and what they aim to test. The manual test suite used in Case 1 contains several quite large test cases, test cases that contain loops and tests to verify the safety requirements of the system, i.e. non-functional attributes of the system. The looping test cases, for instance, aimed to verify the functionality of all the buttons of the GUI, i.e. the same test scenario applied to every button, grouped together into one test case to save space. Performing these tests manually is very time consuming, and tedious, and are therefore often only partially performed, i.e. only a few buttons are tested during test suite execution. However, with the VGT script all of the buttons can be verified during each execution, providing better coverage, without additional cost. However, these tests, even when performed automatically, take quite some time to execute due to the sheer number of buttons that are tested. Thus, since each loop is measurable in length with the test suite’s other test cases but all counted as one single test case, rather than several, it has a large impact on the over execution time of the test suite. Hence, some of the VGT scripts had considerably longer test execution time than others, visualized in a

---

2Manual test execution time was 80 man hours but performed by two testers, i.e. 40 work hours in total.
The slow execution time of the automated scripts in Case 1 can also be contributed to the fact that several of the manual test cases required the tester to just sit and wait for a number of seconds, or even minutes, for an event to be triggered. Tests of this nature aim to test, as an example, the alarm notification service in the tested system, i.e. that an alarm is triggered if the tested system looses its connection to its hardware interfaces. As a more concrete example, if the wind measurement service looses its connection to the airports wind measurement sensors, it should wait for the connection to be re-established within x number of seconds. If the connection is not established within that time, an alarm is triggered. Hence, for x seconds, during this test, the tester is expected to just sit and wait, and since the VGT test cases were implemented in a 1-to-1 fashion they also have to wait. These tests have significant impact on the total execution time of the VGT suite and relate back to the previous statement that the VGT tests are unable to execute tests
5.4. RESULTS AND ANALYSIS

quicker than the tested system can respond. In contrast, the test cases that were automated in Case 2 were more on a functional level and did not consider timing issues, etc, which made it possible for the VGT suite to gain a larger performance advantage over the manual test cases.

However, even though the tests were implemented in a 1-to-1 fashion, results from Case 1 and Case 2 showed that the VGT suites had measurably improved execution time compared to the manual test suite execution time. One factor that explains this speedup is that a human tester continuously has to read the manual test steps in order to know what to input and what output to expect from the system. This factor also explains why test experts can execute the test cases faster, since they are familiar with both the tested system and tests and therefore do not need to spend as much overhead time consulting the test specification. However, in contrast to a human tester, the scripts only needs to execute their commands, i.e. removing the reading overhead completely. Additional execution time gains are provided by the scripts ability to quickly paste textual input into the tested system, whilst a human has to write the input. These gains might seem small, but in the overall perspective, i.e. during execution of the entire test suite, these small gains add up and become significant.

Based on the collected metrics, the return on investment (ROI) for the automation can calculated by comparing the development time of the VGT suites and the manual execution time. Once again, these calculated results are not directly comparable between Case 1 and Case 2 if the contexts of these cases are not taken into account. As shown in Table 5.7, the ROI in Case 1 would become positive after 14 executions of the VGT suite, i.e. the cost of executing the manual test suite 14 times equals the implementation cost of the VGT suite. For Case 2, a positive ROI would be reached after 13 executions. However, since there is no cost related to running the VGT suites, and because they can be run at night, they can greatly improve the test frequency and thereby provide daily feedback to the developers [15]. Hence, both VGT suites would reach a positive ROI within one calendar month, if executed every night, including weekends, given that they identify defects. This ROI calculation is however simplified, since it does not take the number of found defects into consideration, nor the cost for maintenance of the VGT suite. In addition, the maintenance costs of a VGT suite, developed in Sikuli, are still unknown, only initial data has been acquired, and therefore a more detailed analysis of these costs is an important subject of future work.

A graph visualizing when positive ROI is reached, conceptually, based on VGT transition cost and cost per manual test suite execution, is shown in Figure 5.8, supported by the model defined by Berner et al., 2005 [15]. This model assumes that the cost of the manual test case execution is linear, and that the maintenance costs are small and almost linear. Thus, positive ROI would be reached when the two lines cross, i.e. at time tN in Figure 5.8. It should also be noted that during the VGT transition period, t0 to t1, the VGT suite is not executed, whilst the manual test suite could be.
5.5 Discussion

The following sections discuss the findings from this study, their implications for industry, future work, as well as threats to the validity.

5.5.1 Challenges, Problems, Limitations and Solutions

In this report we have presented challenges, problems and limitations (CPLs) identified in two industrial VGT transitioning projects. 58 CPLs were identified during the study, corroborated by information from both projects. They related to different aspects of the transition or usage of VGT, which we divided into three main groups: problems with the tool itself, the system under test, and the support software or environment.

However, despite the many CPLs, the industrial practitioners from Case 2 reported that they found VGT to be both cost effective and efficient at finding faults. In addition, they reported that even though the CPLs caused frustration during the transition project they found ways, e.g. practices or technical solutions, to mitigate or solve them. The practitioners also reported that they had not found any manual interaction system interaction in any of the system test cases that they could not automate using VGT. Similarly, in Case 1 we found CPLs that were more problematic than others, but reported that many of them could be solved or mitigated.

Together this indicates that most CPLs are solvable in practice with either general solutions like the ones proposed in this report, or ad hoc solutions that solve the CPL in the context where the VGT transition project is performed, e.g. by choosing support software that best fit VGT and the tested SUT. In addition, even though the CPLs had negative implications for the transition or later use of VGT; most of the CPLs were perceived as manageable, which is an important result for future work regarding VGT's long-term industrial applicability. However, before any conclusion can be drawn regarding VGT’s long-term applicability, more work is required to determine other aspects of VGT, e.g. the maintenance costs related to the technique. High
maintenance costs has been a problem in previous, similar, GUI-based test techniques. Thus, even though VGT is perceived to be able to mitigate many of the limitations of previous techniques, it is not certain that the costs related to VGT are less, which warrants the need for this research.

However, our study also uncovered CPLs that could not be solved because they required process changes at the studied companies, technical improvements of the tool, etc., which required too great an effort to be applicable in this study. What the impact of these unsolved CPLs will have on the developed VGT suites in the future is unknown and therefore a subject for future work. A hypothesis is however that many of the unsolved CPLs will be resolved as the tools, companies and tested systems evolve, either through technical or process development. Furthermore, since several of the unsolved CPLs were related to the companies processes, e.g. lacking documentation practices, and the tested system, e.g. defects in the system, they are perceived to affect any automation technique, i.e. not just VGT. Thus, contributing to the general body of knowledge regarding automated testing.

Furthermore, several CPLs were identified as VGT tool specific, i.e. related to Sikuli, but in contrast to the test system related CPLs, the Sikuli CPLs were reported to be solvable using different practices, e.g. adding redundant failure mitigation in the scripts or running the scripts from commandline rather than the tool’s IDE. However, even though most of the tool-related CPLs could be mitigated, some still need to be addressed in the future by further development of the tool, for instance to improve the tool’s robustness, make the image recognition algorithm more deterministic, improve the tool’s documentation, etc. The tool-related CPLs are of particular importance since they are the most criticized by industrial practitioners, both at Saab and in other companies that have tried Sikuli during our research. Examples of CPLs that have been highly criticized are the tool’s unstable IDE, its requirement of what version of Java is installed, the unpredictable behavior of the image recognition algorithm, the limitations of the tool’s API, etc. The implications of these CPLs are therefore that many companies, which could have benefited from the technique, are discouraged from using it. Even though many companies state both a need and want for the technique [56].

However, Sikuli is not the only available VGT tool. In our previous work we have compared Sikuli with an anonymous commercial tool and the commercial tool JAutomate [17, 56]. The results showed that the different tools had different properties that also affect their applicability in different contexts. For instance, whilst Sikuli uses the Python programming language for its scripts, JAutomate has a multi-faceted script environment that has been tailored for both novice and expert users. Thus, Sikuli might be appealing to developers that perform testing, whilst JAutomate is more appealing for testers without programming experience. The tool’s different properties also suggests that there might be different CPLs related to different tools, which therefore warrant further studies that replicate the work presented in this report with other tools and in other contexts.

Furthermore, VGT cannot replace manual testing, because VGT scripts, like any other scenario-based script technique, can only find faults specified in said scenarios. Hence, the technique cannot replace an experienced tester’s knowledge how to provoke failures and detect defects [15, 102] In addition,
even though the technique is perceived to be industrially applicable, there are still several important gaps to cover in the technique’s body of knowledge, e.g. empirical evidence to support that VGT’s maintenance costs are feasible, before a conclusion can be drawn regarding its long-term applicability.

### 5.5.2 Defects and performance

In practice, companies are under continuous time-pressure to deliver the software to the customer, which has negative effects on development and quality assurance processes, resulting in software that is likely to contain defects. In the past, one way to uncover these defects has been to use manual high-level practices, which are considered tedious, costly and error-prone [1–6]. High-level automated techniques do exist, e.g. record and replay, but studies have shown that they suffer from several limitations [2, 3, 15, 16, 26, 53]. VGT has properties that perceivably mitigate these previous limitations by, for instance, being robust to GUI layout change, applicable to test applications developed in different languages, etc. Previous work on VGT has also provided initial empirical support for the technique’s industrial applicability [17,18]. However, in order for VGT to also be of value in industrial practice the technique has to be able to identify defects in the tested system. In this report we have presented information, from two projects in industry, which show that a VGT suite is not only able to identify all the defects the manual test suite can identify, but also new ones. These new defects were found because VGT scripts can be executed more cost-effectively compared to manual tests, i.e. potentially every night at minimal cost, with different order between test cases which explores more, potentially faulty, states of the tested system. In combination with the automated high-level interaction the technique can uncover defects that would otherwise have required manual testing, e.g. with new scenarios or exploratory testing, to find. In addition, the execution speed of a VGT suite allows the suite to be executed daily, whilst the industrial norm for manual system tests is once a month to once a year for complex systems. Hence, a VGT suite can provide feedback to the developers about system-level faults with much higher frequency, which is a requested feature of any automation technique [15]. In addition, due to the low cost of running the VGT suite, it can be rerun several times and execute the individual test cases in different order, thereby cover more meta-level scenarios in the SUT. However, the effects of randomized test case execution is still a subject of future work.

In our previous work, we automated approximately 10 percent of the test cases for another version of the tested system used in Case 1 [17]. Based on the collected data from our previous work, we estimated that all the test cases would take roughly 160 hours to automate. However, based on this new data we can refine the estimate for transitioning the full test suite to 426 hours. This large increase comes from the additional requirements and needs uncovered in this larger study, for continuously running automated test suites the test cases needs to be more robust and there needs to be several layers of redundancy and retries to rule out spurious failures that may not depend on the system under test but on the tool or improper setup of the test environment etc. The estimate has also increased due to a skewed distribution in difficulty where a few unexpectedly problematic test cases was found in Case 1. These prob-
lematic test cases for instance included loops, which required more advanced script logic than simple scenarios that started with test step 1 and ended with n. Other problematic test cases required case specific solutions to be found, for instance how to input the Swedish letters å, ä and ö into the Windows login prompt without using Sikuli's type or paste methods. Furthermore, the estimated improved execution time between manual and automated test cases was also incorrect in our previous work. This faulty estimation can once again be contributed to the complex and time consuming test cases that were not covered in our initial study, e.g. test cases that need extensive looping, and test cases requiring the user to just sit and wait for several minutes for a service to timeout. However, based on information collected in Case 2, we found a execution speed-up of a factor 16, compared with the marginal speed-up of 1.06 in Case 1. As reported, the manual test suite in Case 2 only contained smaller test cases, defined in coupled use cases to form longer test scenarios. A conclusion can therefore be drawn that the overall performance increase of a VGT suite is not just dependent on the performance of the SUT, mentioned in Section 5.4.6, but also related to the structure of the manual test specification and how they are carried over into an architecture of automated tests. Hence, in order to gain the highest performance possible from a VGT suite for a legacy system, one should be selective of which, and in what order, test cases are automated, e.g. not automating test cases that require longer waits or loops first.

Overall we conclude that estimating the time for transitioning a large system test suite to VGT is not an easy task, even though it is likely to be a crucial task in industrial decisions on whether and which test cases to automate. Industrial practitioners should try to sample a diverse set of test cases when doing prototype transitions to get better input to the cost estimation process. Our results also indicate that one should consider using a multiplicative factor of between 2-4 times the bare bone estimates if the test environment cannot be fully controlled or if there VGT tool being used will have to be adapted to the company and their systems.

5.5.3 Threats to validity

This case study is limited in that only two projects are considered, both in companies working with safety-critical systems of a similar nature and that have been developed and maintained during several years. This is a threat to external validity. In particular the results might not hold for smaller systems or companies that are young or immature in their development practices and processes. However, the companies use fairly different development processes, one more plan-driven and one more agile, but we have found no evidence that this has any large effects on the transitioning to automated test scripts. There was also a difference in the architecture and structure of the test suite at the two companies. We have mitigated this threat by analysing and discussing this difference in detail. Since both projects used the same VGT tool (Sikuli) the results are likely to depend on the tool. However, in earlier research we found that Sikuli had comparable properties and capabilities to a commercial VGT tool [17] so we think this threat is not a major one.

We still caution that the collected quantitative information should not be
compared without considering the context of these transitioning projects since the projects were performed with different systems, by individuals with different experience and with different information acquisition processes. However, since data collection procedures differed between projects but still corroborated each other we consider little negative effect from these factors on the identified CPL's; apart from the system-specific CPL's they are likely to be seen also in other contexts. In particular, since similar results was seen in the projects even though the data collection was very different between projects.

The researcher doing data collection in Case 1 and the industrial practitioner in Case 2 were all inexperienced with VGT and the tool at the start of the projects. We do not consider this a major threat to validity since this is likely to be typical of these types of transition projects in industry. The more experienced researcher can be considered as a kind of expert consultants which would typically be made available in most companies deciding to do large test suite transitioning. They supported data collection and lead study design in early stages. Also, the experienced researchers lead interviews and feedback workshops during the later stages of the project and performed the analysis. Triangulation was also achieved through independent interviews and inclusion of different roles.

There is additional threats to validity in Case 2 since it was driven by industrial practitioners and the researchers were not on site other than on specific occasions. However, the researchers had continuous contact with the practitioners and focused on setting up good procedures and communication in early stages, as well as collecting experiences and feedback in later stages. We argue that the threats to internal validity that comes from using industrial practitioners in research are unavoidable. If the empirical software engineering community want data from real, industrial projects a certain lack of control must be accepted. Furthermore, since the industrial practitioners had been tasked with doing the transition and evaluating VGT they were highly motivated to do the work and collect relevant data.

5.6 Conclusions

Software industry is faced with challenges that has created a need for research into high-level test automation, a need that Visual GUI Testing, even though it has many challenges, problems and limitations (CPLs), can potentially help solve.

Many companies in current software industry rely on manual test practices to perform high-level testing, e.g. system and acceptance testing, even though these practices are considered costly, tedious and error prone. Automated test techniques, e.g. unit testing and record and replay, have been proposed as solutions to these problems. However, even though there is work to support the usability of these techniques, there is also empirical evidence to suggest that the techniques have problems that limit their industrial applicability in different contexts. Because of these limitations there is still a need for further research into high-level test automation.

Visual GUI Testing is a technique that is emerging in industrial practice which combines image recognition with scripting to automate high-level tests.
Empirical studies have shown the technique’s industrial applicability but like any other technique, VGT has challenges, problems and limitations (CPLs). CPLs that have previously not been explored, leaving a gap in VGT’s body of knowledge.

In this paper we have presented an empirical study performed in two industrial projects where researchers and industrial practitioners used VGT to automate industrial high-level test cases. During the study, 58 CPLs were identified in total that were categorized into 29 mutually exclusive groups of CPLs that relate either to the transition to, or usage of, VGT in industrial practice. The CPLs were further categorized into eight more generic groups of CPLs that relate to the version of the tested system, the tested system in general, defects in the tested system, the company’s processes, the test environment, the VGT tool, the VGT suite or third party software. Further analysis showed that 34 out of the 58 CPLs related to the tested system, the company or the simulator environment. CPLs such as lacking system functionality, misaligned system and test specifications and missing simulator support. Furthermore, 10 CPLs were related to third party software, such as the Virtual Network Connection (VNC) application and screen-recorder software, e.g. VNC lowered the VGT suite success-rate and the recording software wouldn’t start. However, the CPLs with the largest impact during the study were 14 identified tool-related CPLs, e.g. unstable tool IDE and unpredictable image recognition behavior, which caused both confusion and frustration among the study participants. The perceived implications of these CPLs are that industrial practitioners may be discouraged from using, or even trying, the technique, whilst also posing concerns for the long-term applicability of VGT in industrial practice, which is still a subject of future work.

Furthermore, the study also identified four generic solutions that would address about half of the identified CPLs, and mitigate their negative effects. Corroborating results from the two projects also indicated that context specific solutions could be found to most CPLs, e.g. development practices or script logic that mitigated the effects of the tool, test system or support software related CPLs. However, in terms of more general solutions there is still a need for future work, both including technical, e.g. further tool development, and process-oriented solutions, e.g. coding standards.

In addition, the results showed that the researcher who did the data collection and the industrial practitioners found VGT to be both cost-effective and efficient at finding faults despite the CPLs. For instance supported by results from Case 2 where system test frequency was increased from once every six months to several times a week at minimal cost whilst also during the project uncovering three previously unknown faults. Similar results were acquired from Case 1 where four previously known and two unknown faults were identified and reported to the company and corrected. These statements were further supported by the quantitative information that was acquired during the study that indicate that the VGT transition costs are feasible with the potential to provide positive return on investment within one month after development, with up to 16 times quicker execution speed compared to manual tests, whilst still providing equal or even better fault finding ability than manual testing.

In conclusion, this study has shown that VGT is a valuable and cost-
effective technique for high-level test automation but also that it has many CPLs that warrant future research.

Acknowledgment

The authors of this paper would like to thank Saab AB for their participation in these projects and their continued support in answering the question if Visual GUI Testing is an industrially applicable technique.
Chapter 6

Paper E

JAutomate: a Tool for System- and Acceptance-test Automation

E. Alégroth, M. Nass, H. H. Olsson

Accepted at the 6th International Conference on Software Testing, Verification and Validation (ICST’2013), Luxembourg, March 18-22, 2013.
Abstract

System- and acceptance-testing is primarily performed with manual practices in current software industry. However, these practices have several issues, e.g. they are tedious, error prone and time consuming with costs up towards 40 percent of the total development cost. Automated test techniques have been proposed as a solution to mitigate these issues, but they generally approach testing from a lower level of system abstraction, leaving a gap for a flexible, high system-level test automation technique/tool. In this paper we present JAutomate, a Visual GUI Testing (VGT) tool that fills this gap by combining image recognition with record and replay functionality for high system-level test automation performed through the system under test’s graphical user interface. We present the tool, its benefits compared to other similar techniques and manual testing. In addition, we compare JAutomate with two other VGT tools based on their static properties. Finally, we present the results from a survey with industrial practitioners that identifies test-related problems that industry is currently facing and discuss how JAutomate can solve or mitigate these problems.
6.1 Introduction

Manual testing is currently the primary approach in industry to perform system- and acceptance-testing. These tests are performed through interaction with the system under test’s (SUT) graphical user interface (GUI), on a regular basis, to ensure SUT conformance to the requirements, i.e. regression testing [80]. However, manual tests are error prone, tedious and time consuming, known to take up to 40 percent of the entire development budget of a project [105]. To mitigate these problems, test automation has been proposed as the solution, i.e. unit testing [10], record and replay [5, 12, 85], etc. However, these automation techniques have different limitations that restrict their use for system- and acceptance test automation, or make the automated tests costly to maintain. Hence, leaving a gap for a simple, high-level, test automation technique for system- and acceptance-testing.

Visual GUI Testing (VGT), as referred to by Börjesson and Feldt [17], is a tool driven technique that is perceived to cover this gap. VGT combines image recognition with scripts, for automated, scenario-driven, testing through a SUT’s GUI, which allows the user to automate tests on a high level of system abstraction and emulate end-user behavior. The technique is emerging in industrial practice, and empirical evaluation of VGT tools on industrial software has shown that the technique is industrially applicable [17]. However, the tools evaluated in this research required each script to be written manually, raising cost and the number of defects in the scripts.

In this paper, we present JAutomate, a VGT tool that mitigates the need for writing scripts by combining image recognition with record and replay functionality for quick, robust and flexible script development and playback. The paper presents the tool, its general properties, detailed properties of one of the tool’s image recognition algorithms, and a comparison of these properties to two other VGT tools based on information from previous research [17]. In addition, we perform an in depth multi-aspect analysis of the tools impact on a company’s business, architecture, process and organization (BAPO) [106]. The BAPO aspects were chosen to provide breath and depth to the analysis and show how the introduction of the tool will not just affect the adopting company’s test process but also other aspects of the company. Furthermore, we discuss the potential benefits of using JAutomate compared to manual system- and acceptance-testing and present the results from a survey, performed with 52 industrial practitioners, that shows not only an industrial need for the tool, but also that there are test-related problems in industry that the practitioners perceive can be solved, or mitigated, with the tool.

The specific contributions of this paper are:

[A] A presentation of the VGT tool JAutomate, its properties and benefits compared with other VGT tools and manual testing.


[C] A presentation of current industrial test-related problems and discussion of how JAutomate, and VGT, can solve, or mitigate, these problems.

The continuation of this paper is structured as follows. In Section 6.2, related work is presented, followed in Section 6.3 with an in depth presentation,
6.2 Related Work

System- and acceptance-testing is generally performed as a manual practice in industry due to the complexity and high level of system abstraction of the tests. The tests aim to test system conformity to the system requirements, and are performed regularly on the system under test (SUT), i.e. for regression testing [80]. Furthermore, the tests are defined in test specifications, which look different in different companies but with the same basic architecture based on scenarios defined by test steps, i.e. steps that define user input to the SUT for which there is some predefined output. However, this manual test practice is costly, time consuming, tedious and also error prone, requiring resources such as reference systems, testers, etc. Automated testing has been proposed as the solution to these problems, e.g. unit testing [10,34], model based testing [39], etc.

Unit testing [10,34], is a common test automation technique in industry which is performed on the SUT’s component level, i.e. to test low-level functionality of the SUT. This technique is therefore limited and unsuitable, due to complexity and maintenance costs, to use for higher-level tests, i.e. system- and acceptance-tests. These limitations can be mitigated using another commonly used technique in industry, i.e. record and replay (R&R) [5,12,85]. R&R is a tool-driven, two step, technique where user interaction with the SUT is first recorded in a script that can then be replayed automatically to perform regression testing. The recording can be done in several ways, e.g. using references to the SUT’s backend or by using exact coordinates on the SUT’s GUI. However, both of these methods suffer from limitations that, once again, require high maintenance, e.g. reference based R&R is fragile to API or even code change [53], whilst coordinate based R&R is fragile to GUI layout change. Hence, R&R does not fulfill all of industry’s needs for a robust, flexible, high system-level, automated test technique.

In the early 90s, Potter presented his tool Triggers [19], for system automation using image recognition. Other early work on automation using image recognition was performed by Zettelmoyer and Amant [54]. However, in recent years, the use of image recognition has also been transferred to testing, in tools such as JAutomate, Sikuli [87], etc. These tools use image recognition and scenario based scripts to perform tests through the SUT’s GUI, a technique that Börjesson and Feldt refer to as Visual GUI Testing (VGT) [17]. In their work, they provide support for VGT’s industrial applicability by comparing two VGT tools based on their static properties and the tool’s ability to automate manual, industrial grade, test cases for a safety-critical air traffic management system. VGT is perceived to resolve many of the limitations of previous techniques, i.e. R&R, because the technique is black-box [14], i.e. does not require any knowledge about the SUT. Additionally, it is robust to GUI layout change due to the image recognition. However, most VGT tools...
require the test scripts to be written manually, which is associated with an up-front investment for the VGT transition, i.e. the test case automation. Consequently, VGT is perceived to be more flexible and robust than previous automation techniques, but the script development requires an up-front cost.

In this paper, we present JAutomate, a VGT tool that combines image recognition with record and replay capabilities. These capabilities perceivably lower the automation costs, since scripts can be recorded during regular manual test case execution, but retains all the benefits of the image recognition based playback, e.g. imperviousness to GUI layout change. Changes to the graphics of the GUI are instead what imposes the most amount of script maintenance. However, this maintenance can be done at low cost in JAutomate which supports simple swapping of images within the scripts. Hence, JAutomate fills the current gap in industry for a high-level, cost-effective, flexible and robust test automation tool.

However, the introduction of JAutomate, like any new technique, method or tool, in a company, will affect several aspects of said company. There are several frameworks that capture these aspects, e.g. BAPO [106] and PESTEL [107]. BAPO, which stands for business, architecture, process and organization, was chosen as the framework for the analysis in this study because it provides a comprehensive high-level view of what will be affected during software development change in a company. Hence, the introduction of JAutomate in a company will potentially not just affect the testing process, but also the company’s business, e.g. raised quality can be used as a business advantage, architecture, e.g. architectural changes may be required to apply the tool, and organization, e.g. new roles may be required. Thus, BAPO was used to give both a broader, and deeper, analysis of the impact JAutomate can/will have at a company that chooses to adopt it for VGT.

### 6.3 JAutomate

JAutomate is a commercial Visual GUI Testing (VGT) tool developed by Innovative Tool Solutions in collaboration with the Swedish test consultant company Inceptive AB. The tool was innovated by Michel Nass, co-author of this paper, in 2006, after recognizing the potential of the VGT technique and that there were no tools adopting this technique available on the market. JAutomate was in 2006 presented as a concept to the leading test tool vendor at that time, Mercury (Today Hewlet Packard (HP), but no interest was shown by the company. In 2011 the first version of JAutomate was released to the market and has since then been utilized in several industrial projects, e.g. at Volvo, Siemens, CompuGroup Medical. Figure 6.1 shows a screenshot of JAutomate’s Integrated Development Environment (IDE). The IDE is used to develop, i.e. record, execute, i.e. replay, and maintain test scripts.

JAutomate, as mentioned, is however not the only VGT tool available to the market, both open source and commercial alternatives exist. So what makes JAutomate preferable over the other available tools? In the following sections we aim to answer this question by first presenting a comparison of the static properties of JAutomate and two other VGT tools, i.e. Sikuli and CommercialTool (that for legal purposes will be kept anonymous in this report),
6.3. JAUTOMATE

Figure 6.1: A screenshot of JAutomate’s Integrated Development Environment.

results shown in Table 6.1. The information for this comparison was acquired from our previous work [17] and complimented with information provided by JAutomate’s developer, Michel Nass. Secondly, we present a description of JAutomate’s perceived impact on the BAPO aspects of a company [106], based on previous academic empirical work with VGT, and Michel Nass’s expert experiences with JAutomate in industry.

6.3.1 Tool comparison

The following section presents a comparison between JAutomate, CommercialTool and Sikuli based on the tools’ static properties, results summarized in Table 6.1.

Developed in. JAutomate, similar to Sikuli is developed in Java. However, in comparison to Sikuli, JAutomate does not include any native methods which makes it platform independent, i.e. it can be executed on any operating system that supports Java. In contrast, CommercialTool is developed in C# and is therefore only, by the tool’s vendor, supplied for Windows and MacOs.

Image recognition algorithm. This property was not evaluated in detail during our previous work and is therefore unknown on a detailed level for CommercialTool and Sikuli. However, JAutomate uses two algorithms, one based on color and the other on contrast, combined into the so called Vizion Engine. The benefit of having several algorithms, also supported in CommercialTool, is that it adds script robustness. Hence, if one algorithm fails, another can be used instead.

Script language syntax. The scripting language in CommercialTool is custom, based on natural language to make it intuitive for novice users. In Sikuli, the scripting language is based on Python, allowing the user to make use of all the aspects of Python, including iterative statements, conditional branching,
etc. JAutomate, in turn, provides the user with a multi-level scripting interface which on the top layer is designed for novice users without programming experience, whilst on the lower levels, it allows the user to change all aspects of the script through Java code.

*Image representation in tool IDE.* Sikuli and JAutomate both visualize the sought images in the scripts, whilst CommericalTool represents the images as text, i.e. string variables. The benefit of using the actual images is that it makes the scripts more intuitive, but with the drawback, in Sikuli, that the same image might be used in several places and therefore has to be changed in several places during maintenance. However, in JAutomate, this problem has been solved by the use of reusable images, a feature that allows the user to update all images, in all impacted scripts, in a test suite by simply replacing the reusable image (Supported in JAutomate version 11.1 and forward).

*Image recognition sweeps per second.* This metric is dependent on the computer the tool is executed on, but as reported by the official documentation of Sikuli and CommercialTool the tools can make approximately 5 and 7 sweeps per second respectively. No such metric is available for JAutomate since the image recognition algorithms are not only dependent on the performance, i.e. clock rate, of the CPU, but also the number of cores in the CPU, i.e. the algorithms’ performance scales linearly with the number of cores.

*Image recognition failure mitigation.* The image recognition algorithms of all three tools are quite robust, but in some instances they have been known to fail. To mitigate such failure, both JAutomate and CommercialTool deploy a solution with several redundant image recognition algorithms. Hence, if one algorithm fails, the other(s) are used instead. In Sikuli all redundancy has to be built into the scripts manually, e.g. by using exception handling.

*Test suite support.* Both JAutomate and CommercialTool have built in support to build test suites, linking scripts together into more advanced test structures, etc. However, in Sikuli no test suite support exists, only support for individual unit tests. Hence, in order to develop a test suite of Sikuli scripts a custom test suite solution has to be developed, e.g. using Python’s ability to import scripts into other scripts.

*Remote connection support and requirement.* Neither JAutomate or Sikuli has support to control remote computers, e.g. over VNC or remote desktop. CommercialTool in contrast does not only support such a feature, but requires it. Hence, in order to execute scripts locally, with CommercialTool, a virtual machine has to be set up. The benefit of remote test script execution is that it separates the performance intensive execution of the image recognition algorithm from the SUT execution, i.e. mitigating the risk of incorrect SUT behavior due to lack of performance resources.

*Cost.* Sikuli is an open source product, meaning that it is free of charge. In contrast, CommercialTool and JAutomate Studio, i.e. JAutomate’s integrated development environment, are both commercial products that cost 10,000 euros per license per year and $849 per computer per year respectively. JAutomate’s runtime environment is however free of charge and can be used with a regular text editor together with a screen capture tool such as Snagit, etc.

*Record and Replay functionality.* Whilst both CommercialTool and Sikuli require the user to manually input the test scripts, JAutomate supports au-
<table>
<thead>
<tr>
<th>Property</th>
<th>JAutomate</th>
<th>CommercialTool</th>
<th>Sikuli</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed in</td>
<td>Java</td>
<td>C#</td>
<td>Jython</td>
</tr>
<tr>
<td>Image recognition algorithm</td>
<td>Vizion Engine</td>
<td>Several algorithms</td>
<td>-</td>
</tr>
<tr>
<td>Script language syntax</td>
<td>Custom</td>
<td>Custom</td>
<td>Python</td>
</tr>
<tr>
<td>Image representation in tool IDE</td>
<td>Text strings / images</td>
<td>Text-Strings</td>
<td>Images</td>
</tr>
<tr>
<td>Image recognition sweeps per second</td>
<td>Depends on CPU(s) and image size</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Image recognition failure mitigation</td>
<td>Can automatically select from two different algorithms (color and contrast) or perform semi-automated test steps</td>
<td>Multiple algorithms to choose from</td>
<td>Image similarity configuration</td>
</tr>
<tr>
<td>Test suite support</td>
<td>Yes</td>
<td>Yes</td>
<td>Unit tests only</td>
</tr>
<tr>
<td>Remote SUT connection support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Remote SUT connection requirement</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cost</td>
<td>$849 per user per year</td>
<td>10.000 Euros per license per computer</td>
<td>Free</td>
</tr>
<tr>
<td>Record and Replay functionality</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Manual test step redundancy</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Semi-automated test steps</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Backwards compatibility</td>
<td>Yes (based on standard text files and png images only)</td>
<td>Guaranteed</td>
<td>Uncertain</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of the comparison of JAutomate, CommercialTool and Sikuli’s static properties.
automatic recording of the test scripts. This functionality lowers the implement-
tation time of the scripts, especially for scripts that are developed as 1-to-1
mappings to the manual test cases since these scripts can be recorded during
normal execution of the manual test cases.

Manual test step redundancy and semi-automated test steps. Script failure
mitigation due to an identified bug, faults in the script, etc., have already
been discussed. However, JAutomate also has built in support during failure
to, as a last resort, ask the user to perform a test step manually. Hence, if the
image recognition cannot resolve a failure during execution it prompts the user,
through a pop-up, to manually resolve the failure. In addition, JAutomate
includes the possibility of adding manual test steps in the automated execution,
i.e. making it possible to include test steps that require physical interaction
with hardware, e.g. turning on a printer.

Backwards compatibility. Both JAutomate and CommercialTool are guar-
anteed to be backwards compatible. However, Sikuli, being open-source, may
change its instruction set. In previous releases of Sikuli, i.e. version rc2 to rc3,
several methods were changed and new ones introduced. The old methods still
worked, but the recommendation from the developers was to migrate to the
new methods.

Consequently, there are clear differences between the three tools that also
affect their suitability and impact in different contexts. The context impact of
JAutomate on the BAPO aspects is discussed further in the following sections.

6.3.2 Business

From the business aspect of a company, JAutomate, and VGT, will have sev-
eral potential effects, even though they could be considered secondary effects.
The purpose of JAutomate is to provide the company with automated system-
and acceptance-regression tests, which allows the company to raise regression
test frequency. Hence, provide the developers with quicker feedback, which
will lower lead-times and thereby potentially lower development time and raise
quality. Lower development time means lower cost for the customer, and raised
quality that the customer receives a better product, which are both potential
market advantages.

Furthermore, due to the record functionality of JAutomate, the tool could
be used during manual acceptance testing with the customer to capture how
the end-user will actually work with the system. Hence, capturing how the
system will be used in the real-world and thereby improving the tests and raise
the quality of the system.

Consequently, JAutomate has properties that affects a company’s business
through lowering development cost and raising quality, which will provide the
customer with a cheaper, yet higher quality, end product, i.e. qualities that
can give the development company a market edge.

6.3.3 Architecture

The architectural aspect can be viewed from two perspectives. First, the per-
spective of JAutomate, i.e. the tool itself, and second, from the perspective of
the system under test (SUT). In the following sections, these two perspectives
6.3. JAUTOMATE

Figure 6.2: Four images presenting how Vizion Engine derives the contrasts within an image to find a match when performing image recognition.

are discussed together with the benefits and drawbacks of different architectures.

6.3.3.1 The tool

The core of JAUTOMATE, like most VGT tools, is its image recognition capabilities. In order to make the tool as fast and fault tolerant as possible, JAUTOMATE has two image recognition algorithms that are combined into an artifact called the Vizion Engine. The first algorithm identifies images on the screen by comparing pixel color, i.e. comparison between the colors of the image in the script and the image shown on the SUT’s GUI. This approach makes the algorithm fast but less reliable since it makes the algorithm sensitive to the actual color of a widget, which might change dependent on what state the widget is in. For example, the algorithm can not detect an icon, on the Windows operating system (OS), that was unselected during recording if it is selected during playback since the OS tints the icon blue in this state. The second algorithm uses image contrast to identify images, which is slower than the first approach but more reliable. Additionally, these two approaches, combined, makes it possible to also identify images with inverted colors. Images with transparent backgrounds can also be detected.

Figure 6.2 shows two different instances of how the Vizion Engine uses contrast to identify an image on the SUT’s GUI. To the left in Figure 6.2, a picture of a bird is shown as it would be displayed for the user. The picture second from the left, in Figure 6.2, shows how the Vizion Engine views the image after contrast analysis. Hence, the contours of the bird are extrapolated by identifying the largest contrasts in the image, i.e. the difference between the blue bird and the white background. The images to the right of Figure 6.2 shows another example with an image with more details. Using contrast to extrapolate details of the sought images in the SUT GUI has several benefits. First, it lowers the amount of information that is required to find a match, i.e. less comparisons need to be performed. Second, it is more robust, since the algorithm becomes less sensitive to color change, e.g. even images with inverted colors can be identified.

As mentioned, JAUTOMATE, unlike most other VGT tools, supports script recording, a feature that has been added to the tool’s IDE, JAUTOMATE Studio, to mitigate the costs related to test development and maintenance. Recordings are performed with JAUTOMATE’s recorder widget, shown in Figure 6.3. When the record button, shown to the left of Figure 6.3, is pressed, JAUTOMATE automatically records all user interaction, e.g. mouse and keyboard events, with the SUT’s GUI. When the stop button, shown to the right of Figure 6.3, is pressed, the interactions are compiled into a script that can then
be replayed automatically. However, the scripts are seldom perfect after the initial recording and it is therefore possible to manually adjust and modify the scripts post-recording to improve captured images, add delays to better synchronize the script with the SUT, etc. In addition, to improve script quality, JAutomate contains automated script adjustment support, i.e. the tool can automatically improve captured images based on new information acquired during execution of the script. Furthermore, scripts in JAutomate Studio consist of a custom scripting language which has been designed to be intuitive to users without programming experience. However, for more advanced users, the architecture of the tool allows the user to manipulate the scripts on several levels of abstraction, from the top custom script language down to a core level which consists of Java code. Additionally, for advanced users, JAutomate provides features to tailor the high level scripts, e.g. manipulation of test script parameters, test suite support, conditional test execution and iterations. For instance, JAutomate supports data driven testing where the input parameters are provided by a comma separated text file. This feature lowers development cost since it allows the tool to perform the same test scenario over and over, but with different input- and output-parameters, instead of requiring one script per input and output. In addition, the tool supports creation of suites of tests that can either be executed in sequence or build on one another to form more advanced tree structured test hierarchies, which promotes modularization and reusability. Conditional test case execution is also supported, which is useful when there are dependencies between test scripts, i.e. the outcome of test case X prohibits the execution of test case Y, requiring test case Z to be executed instead. This feature mitigates the risk of test suite failure due to identification of a bug, image recognition failure or other failure caused by JAutomate or the SUT itself. Consequently, JAutomate is a multi-layered tool that is perceivably appealing to both novice and expert users by having high learnability for simple functionality, but with high degrees of flexibility due to the tool’s advanced functionality.

Additionally, the architecture of the Vizion Engine is designed to scale based on the number cores in the central processing unit (CPU) of the computer executing the tool. Thus, the speed of the algorithms increase linearly with the number of cores in the CPU, which can be observed both during playback and recording of the scripts. Additionally, the tool supports several connected computer monitors, i.e. desktops with higher resolution. However, higher resolution has detrimental effects on the speed of the image recognition algorithm since more area has to be sought through to find a matching image. However, the minimum requirements to execute JAutomate are still quite modest, despite the performance intensive image recognition algorithm, requiring only 128 MB of free Random Access Memory (RAM). For longer scripts 256 MB of RAM is however recommended.

JAutomate is developed in pure Java code, i.e. the tool does not contain
any native code, third party modules or libraries, which makes it platform independent. Hence, the tool can run on any computer as long as a Java run-time environment, Java 6 or a later version, supplied by Oracle, is installed. In addition, as reported by the tool’s developer, the use of pure Java code makes the tool easy to maintain and extend with new functionality. All maintenance of the tool is performed by Innovative Tool Solutions in collaboration with Inceptive which is the tool provider and also the provider of a tool implementation service. Tool support is provided through e-mail by Innovative Tool Solutions or by Inceptive through telephone or personal support (Restricted to customers of JAutomate Studio).

6.3.3.2 The system under test (SUT)

Similar to all VGT tools, JAutomate considers the system under test (SUT) as a black-box [14], meaning that the tool does not require any knowledge about the SUT’s internal architecture, components, development language, etc. The black-box approach is achieved through the high-level interaction of the tool, i.e. through the image recognition that allows the tool to provide input and observe output through the SUT’s GUI. Consequently, all automated tests developed using JAutomate are executed in the same way as a human user would perform them, i.e. through the operating system’s internal methods which makes the tool’s interactions indistinguishable from a human user from the SUT’s perspective. In contrast, previous GUI-based test techniques, e.g. widget based record and replay, interact with the SUT by calling operations/methods within the SUT, i.e. white-box testing [14]. Hence, these interactions are performed below the GUI layer and thereby differentiate from end user interactions. In addition, some white-box tools require hooks, or interfaces, to be added in the SUT, thereby changing the internal architecture of the SUT. Hence, JAutomate, in contrast to white-box tools, is non-intrusive and does not require any changes to the SUT in order to be applicable. Furthermore, due to the GUI-based interaction, the tool can interact with any type of system that can be accessed manually from a computer client, i.e. it can test desktop, web and even mobile applications.

Because of these properties, JAutomate can be used to convert manual test cases into equivalent automated tests, i.e. produce 1-to-1 mapped automated tests from the manual tests. Automated 1-to-1 mapped tests require less resources than their manual equivalents and allow a company to perform system regression tests with higher frequency, providing developers with quicker and more comprehensive feedback. However, depending on the SUT architecture, which can contain hardware components, e.g. devices such as printers, there are manual test steps that are impossible to automate using software test tools, e.g. check if a printer is turned on or physically press an emergency stop button. Therefore, JAutomate supports the insertion of manual test steps in the automated scripts. When a manual test step is triggered, the automated test execution is paused and a pop-up dialog is displayed on the screen with the manual test step instructions. Once the manual test step has been performed by the user, the user can tell the script to continue its execution. In addition, JAutomate supports semi-automated test steps. Hence, if the automatic execution fails, e.g. due to image recognition failure, the user is prompted with a
manual instruction to resolve the problem before the automated test execution can be continued. Semi-automated test steps are useful when testing volatile, or concurrent, systems where the SUT’s behavior is difficult to predict. This feature also helps mitigate failure due to unexpected system behavior such as pop-up windows prompted by the operating system, other applications, etc.

6.3.4 Process

Similar to the introduction of any tool, practice, method, etc., JAutomate will impact the process of the target company, i.e. change how development and testing is performed. However, no empirical study has been conducted with the purpose of identifying what these changes might be or what impact they will have. The design of JAutomate does however allow it to run manual tests, as explained in Section 6.3.3.2, similar to the manual test runner in HP Quality Center. Thus, the tool can be used for creating test suites that are a mix of automated, manual and even semi-automated tests. Additionally, this functionality allows a test team to use JAutomate to define all their system- and acceptance-tests, i.e. not just automated tests. However, we do not propose that JAutomate should replace a company’s other testing tools, methods or practices, rather, we propose that a company should add the tool to their existing toolbox. Hence, JAutomate is not a replacement, but rather a compliment, to other tools and/or manual practices, which provides support for automated system- and acceptance-tests.

In order for JAutomate, and VGT as a technique, to be effective, the tool needs to be incorporated into the company’s testing process and be used in a continuous integration fashion, i.e. the VGT scripts should be executed every time the system is rebuilt. Additionally, a maintenance process needs to be put in place in order to ensure that the automated test suite is continuously up to date with the SUT, similar to the practice of maintaining manual test suites. Consequently, JAutomate will introduce overhead in terms of additional maintenance since both the scripts and the manual tests need to follow the system specification, especially for 1-to-1 mapped tests. However, due to JAutomate’s ability to define manual test steps within the scripts, the manual tests can be migrated into JAutomate test cases, i.e. have the manual test specification defined in JAutomate scripts. A practice that, perceivably, would mitigate the need for maintenance of several equivalent test artifacts.

6.3.5 Organisation

JAutomate is not considered to have a large impact on the organization, e.g. in terms of requiring new roles or redistribution of resources. The reason is because unlike other test techniques for complex testing, that sometimes require test automation specialists or experts to develop tests of high quality, e.g. in machine learning [108], JAutomate has been designed to be simple to use for any user. Thus, JAutomate has been developed to have high usability and learnability to allow both developers and testers with, or without, previous SUT or programming experience to use the tool. Hence, JAutomate does not require any change of a company’s current roles, introduction of new roles or infer replacement of human testers in their current roles. Instead, JAutomate is
6.4. THE INDUSTRIAL NEED

In order to investigate the industrial need for JAutomate, and VGT in general, a survey was performed during a seminar about the technique, attended by approximately 100 industrial practitioners. The survey had two purposes. First to identify the test related problems that Swedish software development companies are currently facing. Second to evaluate the industrial practitioners’ knowledge about VGT, and JAutomate, and if VGT perceivably could solve, or mitigate, some of the test-related problems experienced by the practitioners. The first purpose was investigated through a question where the industrial practitioners got to distribute 100 points to rank 17 predefined test-related problems. 52 questionnaires were collected, hence a response rate of roughly 50 percent. Figure 6.4 visualizes the results from the first question, shown as the percentage of distributed points over the test-related problems.

Figure 6.4: Results of the first survey question, ”What/Which are the largest problem(s) with your company’s current system- and acceptance-testing?” The y-axis shows the questionnaire’s 17 predefined answers and the x-axis shows the answers provided by the industrial practitioners, shown in percent.

primarily a compliment to previous practices to make the testing more efficient in terms of cost and quality. In addition, it is perceived that VGT tools, e.g. JAutomate, help alleviate the tediousness of performing manual tests over and over. Tediousness that lead to developers and testers taking shortcuts, making mistakes, etc., which lowers the quality of the testing, and in extension, software quality. Consequently, the impact on a company’s organization is low but high on the company as a whole.
possible to raise the test frequency. Additionally, the problem with insufficient
customer feedback can be mitigated by recording customer acceptance tests,
mitigating the need for having customers on site for acceptance regression
testing. Problems regarding test coverage can perceivably also be mitigated
since VGT releases resources required for manual testing, i.e. developers and
testers. These resources can instead be used to perform exploratory testing
of the system to uncover previously unknown faults. Hence, VGT, performed
with JAutomate, has the potential to solve or mitigate several of the largest
test related problems encountered in industry.

The remaining questions in the survey concerned the industrial practition-
ners’ knowledge, need and interest in VGT and JAutomate. The practitioners’
knowledge was elicited by asking the practitioners’ if they had heard about
VGT before the seminar where the survey was performed. 73 percent of the
participants said that they had never heard of it, or only heard of it in passing.
The industrial need for the technique was elicited by asking if they perceived
that VGT could solve their current test-related problems. 33 percent answered
that they were certain, 44 percent were uncertain and the remaining 23 per-
cent were doubtful or did not think VGT would help. However, 67 percent of
the practitioners who stated that VGT would not solve their test related prob-
lems also stated that they worked with systems without GUIs, i.e. systems
for which VGT is not applicable. Finally, the interest of VGT was elicited by
asking the practitioners if they would investigate VGT further after the sem-
inar. 71 percent stated that they would, 21 percent were uncertain and the
remaining 8 percent that it was doubtful or that they would not investigate
it further. Consequently, the survey showed that there is both a need and an
interest for the technique in industry.

6.5 Discussion

Visual GUI Testing (VGT), with tools such as JAutomate, have been designed
for development of high system-level tests for automated regression testing.
Automated regression testing is generally proposed to be a good practice and
has been incorporated in agile development processes such as eXtreme Pro-
gramming to facilitate continuous integration (CI) [109]. However, CI testing
is mostly associated with XUnit testing [10], but, as proposed by Fowler [109],
other tools for end-to-end testing should be incorporated into the test process
as well, e.g. FitNesse [110]. JAutomate has been designed with CI in mind
and the test scripts can even be exported and run in FitNesse, or stand alone
as a compliment to other automated testing. Hence, JAutomate is, and was
designed, to be a compliment to other automated testing in a CI context, i.e.
providing support for automated system- and acceptance-testing that previ-
ously had to be performed manually.

Furthermore, this report has presented JAutomate as a testing tool to
perform VGT. However, given the capabilities of the tool, e.g. its ability to
interact with any GUI based application, the tool can also be used for automa-
tion, e.g. to automate the build process performed during CI to minimize the
risk of erroneous builds due to complexity [109]. In addition, JAutomate opens
up new possibilities for monitoring of systems where it is unfeasible to use hu-
mans, e.g. to monitor memory usage during long-time tests or tests where input is given continuously to the SUT for longer periods of time, e.g. for 24 hours straight. Furthermore, JAutomate, and VGT, is perceived to be able to test non-functional properties of a SUT, e.g. usability, performance, etc, but no study has been performed to validate this claim, which is a subject of future research.

This report also presents a comparison between JAutomate and two other VGT tools based on their static properties, results shown in Table 6.1. An analysis of these results show that JAutomate has several benefits compared with the other tools, such as being platform independent, has record functionality for fast script development as well as manual and semi-automated test step execution. The tool also has beneficial features in common with one or both of the other compared tools, such as multiple image recognition algorithms, images within the scripts, comprehensive failure mitigation, test suite support and backwards compatibility. However, the tool lacks built in support for remote SUT connection, supported by CommercialTool. Previous research with Sikuli, corroborated with information collected by Michel Nass at ComputGroup Medical, does however show that VGT tools, including JAutomate, can be executed on top of third party remote SUT connection applications, e.g. virtual network connection (VNC) applications. Remote SUT connection allows VGT tools to test SUTs that are distributed over several computers and removes the performance intensive image recognition execution from the SUT. Thus, mitigating the risk of faulty SUT behavior due to lack of performance resources.

However, the maintenance costs of VGT scripts, e.g. with JAutomate, are still unknown and is therefore the main focus of future research. Empirical support that validates the feasibility of these costs is essential for the long-term applicability of the technique, especially since maintenance costs have been identified as one of the main problems with previous, similar, techniques such as widget based record and replay.

6.6 Conclusion

In this paper we have presented JAutomate, a tool, for Visual GUI Testing (VGT) with record and replay support for cost effective script development, for graphical user interface (GUI) based, automated, system- and acceptance-tests. The tool has several benefits compared to manual regression testing, and other VGT tools, which will impact a company in several aspects. Furthermore, a survey with industrial practitioners, showed that there is both industrial need and interest for the tool and VGT as a technique.

The industrial needs to lower lead times and raise software quality are ever growing, pawing the way for new research into automated techniques and tools for all aspects of software engineering, from requirements engineering to testing. Test related automation techniques approach testing on different levels of system abstraction, from system- to code-level. However, due to different limitations of these techniques and tools there is currently a gap for a high-level, cost-effective, flexible and robust tool for system- and acceptance-test automation.
JAutomate fills this gap, with simple record and replay functionality, combined with image recognition, which allows the user to automate all types of user interaction performed through the system under test’s (SUT) GUI. A comparison between JAutomate and two other VGT tools shows that JAutomate has several benefits over the other tools, but also that the tools have different properties that make them suitable in different contexts. In addition, we have presented how JAutomate will have beneficial impact on other aspects than just the test process. For instance, from a business aspect, JAutomate will help raise software quality, which can be a business advantage. Additionally, since the tool is black-box it does not require any knowledge or modification to the SUT’s architecture in order to be applicable. Furthermore, the tool is designed with both novice and advanced users in mind and does therefore not affect the company’s organization, e.g. by requiring new, or changed, roles.

In addition, a survey performed with industrial practitioners showed that JAutomate, and VGT as a technique, can perceivably help solve many of the software market’s current test-related problems and that there is an industrial interest for the tool and the technique. The tool also opens up new possibilities for testing and automation, e.g. emulation of user monitoring during long-time tests.

Consequently, JAutomate is a promising tool that fills a current need in industry for a flexible, robust, easy, cost-effective, automation tool for GUI-based system- and acceptance-testing, to compliment companies current test toolbox.
Bibliography


