Noname manuscript No.
(will be inserted by the editor)

Visual GUI Testing in Practice: Challenges, Problems and Limitations

Emil Alégroth · Robert Feldt · Lisa Ryrholm

Received: date / Accepted: date

Abstract In today's software development industry, high-level tests such as Graphical User Interface (GUI) based 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 therefore 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 with perceived higher flexibility and robustness to certain GUI changes than previous high-level (GUI) test automation techniques. The core of VGT is image recognition which is 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-based knowledge of these CPLs and how they impact industrial applicability. Crucially, there is also a lack of information on the cost of applying this type of test automation in industry.

This manuscript reports an empirical, multi-unit case study performed at two Swedish companies that develop safety-critical software. It studies their transition from manual system test cases into tests automated with VGT. In total, four different test suites that together include more than 300 high-level system test cases were automated for two multi-million lines of code systems. The results show that the transitioned test cases could find defects in the tested systems and that all applicable test cases could be automated. However, during these transition projects a number of hurdles had to be addressed; a total of 58 different CPLs were identified and then categorized into 26 types. We present these CPL types and an analysis of the implications for the transition to and use of VGT in industrial software development practice. In addition, four high-level solutions 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. Nine of the identified defects are reported, 5 of which 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 related to the transition and usage of VGT, the technique is still valuable, flexible and considered cost-effective by the industrial practitioners. The presented CPLs also provide decision support in the use and advancement of VGT and potentially other automated testing techniques similar to VGT, e.g. Record and Replay.

Keywords Visual GUI Testing · Industrial case study · Challenges, Problems and Limitations · System and acceptance test automation · Development cost

Emil Algroth
Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg
SE-412 96 Gothenburg, Sweden
Tel.: +46703-338022
E-mail: emil.alegroth@chalmers.se

1 Introduction

Currently available automation techniques for high-level testing, i.e. GUI-based 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 [Rafi et al(2012)Rafi, Moses, Petersen, and Mantyla,Gutiérrez et al(2006)Gutiérrez, Escalona, Mejías, and Torres,Berner et al(2005)Berner, Weber, and Keller,Grechanik et al(2009a)Grechanik, Xie, and Fu,Horowitz and Singhera(1993),Sjösten-Andersson and Pareto(2006),Zaraket et al(2012)Zaraket, Masri, Adam, Hammoud, Hamzeh, Farhat, Khamissi, and Noujaim, Finsterwalder(2001)]. These reported drawbacks with current automation techniques is one factor why manual testing is persistently used in industrial practice even though it is considered costly, tedious and error prone [Grechanik et al(2009c)Grechanik, Xie, and Fu,Grechanik et al(2009a)Grechanik, Xie, and Fu,Finsterwalder(2001),Leitner et al(2007)Leitner, Ciupa, Meyer, and Howard,Memon(2002),Dustin et al(1999)Dustin, Rashka, and Paul,Cadar et al(2011)Cadar, Godefroid, Khurshid, Pasareanu, Sen, Tillmann, and Visser].

Manual testing has long been the main approach for testing and quality assurance in 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 [Itkonen and Rautiainen(2005a)]. 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 [Miller and Collins(2001), Hsia et al(1997)Hsia, Kung, and Sell, Hsia et al(1994)Hsia, Gao, Samuel, Kung, Toyoshima, and Chen, Myers et al(2011)Myers, Sandler, and Badgett]. However, software is prone to change and therefore requires regression testing [Onoma et al(1998)Onoma, Tsai, Poonawala, and Suganuma, Rothermel et al(2001)Rothermel, Untch, Chu, and Harrold], 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 [Dustin et al(1999)Dustin, Rashka, and Paul]. Many automated test techniques approach testing from lower levels of system abstraction, e.g. unit tests [Olan(2003), Gamma and Beck(1999),Cheon and Leavens(2006)], which make them a powerful tool for finding component and function level defects. 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 [Berner et al(2005)Berner, Weber, and Keller,Grechanik et al(2009a)Grechanik, Xie, and Fu].

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 [Holmes and Kellogg(2006)], JFCUnit [Hackner and Memon(2008)], TestComplete [Vizulis and Diebelis(2012)], 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 have identified limitations with these techniques. They are typically sensitivity to GUI layout or code change and the tools are dependent on the specifics of the system implementation, etc., which negatively affect the techniques' applicability and raises the cost of maintaining the tests [Horowitz and Singhera(1993), Sjösten-Andersson and Pareto(2006), Grechanik et al(2009b)Grechanik, Xie, and Fu, Berner et al(2005)Berner, Weber, and Keller].

Visual GUI Testing is an emerging technique in industrial practice that combines scripting with image recognition in open source tools such as Sikuli [Yeh et al(2009)Yeh, Chang, and Miller], and commercial tools such as JAutomate [Alegroth et al(2013b)Alegroth, Nass, and Olsson]. Image recognition provides support for emulating user interaction with the bitmap components, e.g. images and buttons, shown to the user on the computer monitor and can therefore enable testing of GUI-based systems, regardless of their 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 defect finding ability than its manual counterparts [Börjesson and Feldt(2012), Alegroth et al(2013a)Alegroth, Feldt, and Olsson]. However, VGT is not without its challenges, problems or limitations (CPLs), such as CPLs related to tool immaturity, image recognition volatility and limitations in use for highly animated GUIs. These CPLs have not been sufficiently explored in existing research 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 the technique. Furthermore, we make no distinction between the three CPL subtypes in this paper, i.e. between challenges, problems and limitations, since a challenge in one context could be a problem in another. As such, we define a CPL *as a challenge, problem or limitation that hinders the transition to or usage of VGT in a given context* *but where no distinction is made between the three CPL subtypes since their impact in different contexts will vary.*

In this paper we present an empirical study performed at two different companies within the Swedish defense and security systems 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. However, due to differences between the study contexts, this manuscript will report both corroborated, generalized, results and context dependent results. Context dependent results are reported together with the factors that affect their generalizability.

During the study, a total of 58 CPLs were identified that were categorized into 26 unique CPL types that affect either the transition to or usage of VGT in industrial practice. These CPL types have implications for the industrial applicability of VGT but also other automated test techniques, e.g. adding frustration, confusion and cost to the transition and usage of the techniques. Consequently, the results of this study contribute to the general body of knowledge on automated testing, which is currently lacking empirical evidence regarding CPLs [Rafi et al(2012)Rafi, Moses, Petersen, and Mantyla]. We also present several CPL types that are present for other test techniques, e.g. how to synchronize scripts and the SUT, common to Record and Replay tools, misalignment between system implementation and specification which affects all testing, etc.

In addition, several solutions were identified during the study that solve most of the identified CPL types. However, since most of these solutions were ad hoc, and the focus of this work is on the VGT CPLs, only four general solutions will be presented. These solutions solve roughly half of the identified CPL types and provide guidance to industrial practitioners that intend to evaluate or use the technique.

Furthermore, quantitative information regarding the VGT suites' defect finding ability, the VGT suites' development costs and the projects' evaluated return on investment are presented. This information can 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.

We also present, based on results from literature, that not only are the identified CPLs and solutions to said CPLs general for other VGT tools, but other test techniques as well. Thus providing a more general contribution of empirical results to the software engineering body of knowledge.

To summarize, the specific contributions of this work for test researchers and industrial practitioners 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 comparison to the cost of manual test suite execution.

The next section, Section 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 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 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.

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 [Myers et al(2011)Myers, Sandler, and Badgett]. 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 [Regnell and Runeson(1998), Memon et al(2003)Memon, Banerjee, and Nagarajan], 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 [Miller and Collins(2001), Hsia et al(1997)Hsia, Kung, and Sell, Hsia et al(1994)Hsia, Gao, Samuel, Kung, Toyoshima, and Chen, Myers et al(2011)Myers, Sandler, and Badgett], 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 [Grechanik et al(2009c)Grechanik, Xie, and Fu, Grechanik et al(2009a)Grechanik, Xie, and Fu, Finsterwalder(2001), Leitner et al(2007)Leitner, Ciupa, Meyer, and Howard, Memon(2002), Dustin et al(1999)Dustin, Rashka, and Paul]. 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 [Lowell and Stell-Smith(2003), Andersson and Bache(2004)].

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. [Adamoli et al(2011)Adamoli, Zaparanuks, Jovic, and Hauswirth]. In their work on automated performance testing they identified 50 articles related to automated GUI testing using different techniques. One of the most common techniques they identified was capture/Record and Replay (R&R) [Adamoli et al(2011)Adamoli, Zaparanuks, Jovic, and Hauswirth, Andersson and Bache(2004), Memon(2002)]. 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 [Memon and Soffa(2003), Zaraket et al(2012)Zaraket, Masri, Adam, Hammoud, Hamzeh, Farhat, Khamissi, and Noujaim, Finsterwalder(2001), Li and Wu(2004), Grechanik et al(2009b)Grechanik, Xie, and Fu, Horowitz and Singhera(1993)]. The coordinate-based R&R techniques are sensitive to GUI layout change [Memon(2002)], 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 [Sjösten-Andersson and Pareto(2006)]. 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 requires more direct access to the SUT, for instance through hooks added into the SUT or access to its GUI library, e.g. Java Swing, PyGTK, Eclipse RCP, etc., which limits the tools' applicability. In addition, these tools can generally only interact with one GUI library at a time and also require special interaction code to be developed to interact with custom GUI components. Some exceptions exist, e.g. TestComplete [smartbear(2013)], which supports 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 [Chang et al(2010)Chang, Yeh, and Miller], and commercial, e.g. JAutomate [Alegroth et al(2013b)Alegroth, Nass, and Olsson], but still 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 [Börjesson and Feldt(2012)], 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 both tools were fully capable of performing the automation with equal defect 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 [Alegroth et al(2013a)Alegroth, Feldt, and Olsson]. 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.

The study presented in this paper builds on our previous work but with another focus. Whilst the previous works have focused on the applicability of VGT, with successful and corroborating results that such is the case, this study focuses on the challenges, problems and limitations of VGT. Thus, information that is currently missing in the academic body of knowledge [Rafi et al(2012)Rafi, Moses, Petersen, and Mantyla]. Hence, this work aims to provide a more balanced view of the benefits and drawbacks of the technique based on empirically collected information. As such, providing both researchers and industrial practitioners with insights as to what hurdles they should focus on in research or expect when transitioning to VGT in practice. Furthermore, we also present the adaptations and architectural solutions that were taken in order for the companies to be successful in their VGT transition. Thus, contributing to research application rather than fundamental research of the technique and its tools. In addition, since the observed CPLs and solutions to said CPLs are identified to be generic for contexts other than VGT, our results and conclusions also provide a more general contribution to the academic body of knowledge regarding automated testing.

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. [Rafi et al(2012)Rafi, Moses, Petersen, and Mantyla] 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.

3 Industrial case study

The empirical study presented in this manuscript consisted of two VGT transition projects. VGT transition/automation is in this study defined as *the practice or process of manually writing or recording a VGT test script that automatically performs GUI-based interactions and assertions on the system under test equivalent to a manual test case for said system under test.* As such, a VGT (test) script is defined

[1] The name of CommercialTool can not be disclosed due to confidentiality reasons.

as *a programming artifact with coded interactions and GUI bitmaps that can be executed to autonomously stimulate a system under test through its GUI for test or automation purposes.*

The first VGT transition project was performed at the company Saab AB, in the Swedish city of Gothenburg, building upon our previous work [Börjesson and Feldt(2012)]. The VGT transition was performed by a research team member with no prior knowledge of Saab or VGT but with frequent support from the research team. Focus was therefore on the applicability of VGT to automate a complete manual test suite for an industrial system and the challenges, problems and limitations (CPLs) of said VGT transition. Applicability is in this context defined as *how well and at what cost a technique/process/practice can be implemented in a given context to provide benefit.* As such, Visual GUI Testing is highly applicable in a context where it requires only small and cheap changes to implement and it gives quick return on investment, whilst it has low applicability in a context where requires large and costly changes to implement with slow return on investment

The second VGT transition project was performed in another Saab company within the Saab corporation, in the Swedish city of Järfälla, and had two focuses. First, to evaluate if VGT could be applied in practice in an industrial project, results presented in [Alegroth et al(2013a)Alegroth, Feldt, and Olsson]. Second, to evaluate the CPLs experienced by industrial practitioners during VGT transition in this context. As such, the second project was driven by industrial practitioners with minimal support of the research team.

The two projects complement each other by providing information of scientific rigor from the first project and information from the practical usage of VGT from the second project. Thus, this study contributes to the academic body of knowledge with the results from two empirical holistic case studies [Runeson and Höst(2009)] from two different companies[2], in the continuation of this manuscript 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 [Runeson and Höst(2009)]. 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.

In the following subsections we first present the industrial projects and their context followed by two sections that give a more in depth view of the companies and how the projects were performed. Finally, in the fourth subsection, we will present the test suite architecture that was used in both companies.

3.1 Industrial projects and Research Design

Both VGT transition projects were conducted in industry with two mature industrial software products/systems. In Case 1, the VGT scripts were developed from a scenario-based test suite intended to be performed manually by testers on a distributed, safety-critical, air traffic management system. The system consisted of an excess of 1 million lines of code developed in several programming languages with the bulk of it written in C# and XML. XML was used to configure the system for different customer needs. Configuration included, but was not limited to, changing the layout of the graphical user interface (GUI), data interface timeouts, the simulator environments, etc.

Furthermore, the system's GUI was shallow, meaning that it consisted of a set of GUI components that were continuously shown to the user with the property that interaction with a specific component did not obstruct or deny interaction with any other component in the GUI. An example of another application, that should be known to the reader, with this GUI property is a calculator application. As such, the GUI view was flat but the system's backend functionality was advanced, i.e. airstrip and airport radar control. The GUI was also distributed across two separate, physical monitors connected to two computers and one server over a local area network. This system attribute required special consideration during the study that will be described in Section 3.2. In the development environment both parts of the GUI were controlled using a mouse and keyboard. However, after deployment at a customer one of the GUI's was instead controlled using a touchscreen monitor. The touchscreen monitor put specific requirements on the size and look and feel of the GUI components.

[2] Even though the companies are part of the same corporation they have no contact in practice and can be considered different organizations

The system has been developed in both plan-driven and iterative development projects that started 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 system's test description that specifies the manual test cases used for system testing. This system testing is typically performed once or twice every development iteration at the company, i.e. once or twice every six months. During such a system test, all test cases in the test description are performed and any behavioral deviations noted. The test cases were defined as scenarios that were documented in tables where each row defines input and expect output for each test step. Because the system's GUI was distributed, most test cases also required the tester to physically move from one computer monitor to the other. A more detailed description of the test specification can be found in Section 3.4.

Furthermore, the system was developed in compliance with RTCA DO-278 quality assurance standard for airplane and air traffic management software. RTCA DO-278 compliance is required by the system's customers that consist of domestic public and military airports as well as international public airports. The standard defines a set of test quality objectives that all test practices/techniques, the company use, must adhere to, including VGT.

Most of the company's developers hold masters of science and technology degrees and have, on average, three to five years of industrial experience as developers, testers and/or project managers. Furthermore, the company does not have dedicated testers, instead all testing is performed by the developers. The company has a hierarchal, yet flexible, organization, situated in two Swedish sites: Gothenburg and Växsjö.

In Case 2 the VGT transition was performed for a battlefield control system that was both safety- and mission-critical. The system was distributed across several computers and tested with test cases that required interaction with several physical computers, which as stated required further consideration that will be explained in Section 3.2.

Due to the limited involvement of the research team in Case 2, but primarily for 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, and is similar in terms of complexity and criticality as the system in Case 1. The system is used for military control and its main feature is to allow military commanders to track the movement of friendly and enemy forces on the battlefield. This feature is accomplished through a sophisticated map engine that allows the commander to drag and drop units onto the screen but also scroll around and scale the map for battlefield awareness and overview. Hence, the system is GUI-driven but with a deeper GUI than in Case 1. The deeper GUI means that interaction with a specific component from the set of all GUI components in the system can obstruct or deny interaction with another component on the GUI. An example of such an application, which should be familiar to the reader, is a webpage with several pages. Hence, the system has several views that, at least partly, cover each other. In the test environment manual interaction with the system is performed using a mouse and keyboard but when deployed, as for the system in Case 1, the system is controlled through a touchscreen monitor.

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, the tested system did not adhere to any specific quality assurance standard. The look and feel of the system's GUI was however specified by a military standard with specific regulations on button sizes and positioning to maximize its usability in a mobile environment, i.e. in a vehicle moving across rough terrain. Finally, the product's main customer is a Swedish military contractor.

Figure 1 visualizes the research process and the three parallel tracks that were performed by the researchers in Case 1, practitioners in Case 2 and the support and data acquisition performed by the research team. The information acquisition and support activities were continuous in Case 1 with members of the research team at Saab daily during the project. The VGT transition was performed by one member of the research team, supported by the other researchers and developers at the company when needed. Information was acquired according to a structured process that will be described in section 3.2.

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. In the first workshop, information such as what type of system they were working with, what the Sikuli test architecture would look like, etc., was acquired. In relation to the second workshop, two semi-structured, one hour, interviews were also 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).

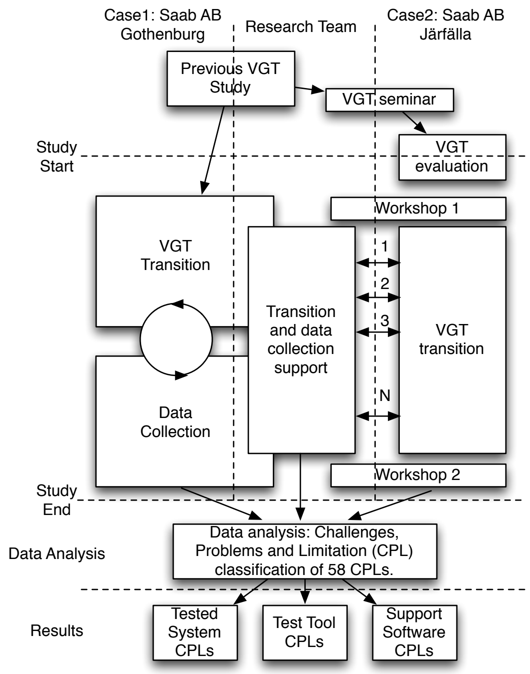

Fig. 1 Visualization of the research process showing three (vertical) tracks. Track 1 (leftmost) contains activities performed in Case 1 and prior work at Saab AB in Gothenburg. Track 2 (middle) the activities of the research team and Track 3 (rightmost) the activities of Saab AB in Järfälla. Boxes that cross over the dotted lines were performed by the research team and the respective company.

After the completion of Case 1 and Case 2, all acquired information was analyzed to identify the CPLs. The input for the analysis was the detailed documentation of observations, the interview transcripts and the email correspondence from Case 2. When a potential CPL was identified through qualitative document analysis of the input information, its characteristics were compared to previously identified CPLs and if the new CPL was found to be unique the CPL would be abstracted and counted. The initial analysis found 58 CPLs. However, many of the CPLs had such similar characteristics that they could be grouped together into types, resulting in a reduction from the 58 identified CPLs to 26 unique types with one to seven CPLs in each type. In order to mitigate bias, this analysis was performed by a member of the research team that had not taken part in the VGT transition. Additionally, to mitigate bias and faulty assumptions made by the analyst, the 26 CPL types were discussed and validated with the participants of the study, i.e. the research team member from Case 1 and the industrial practitioners from Case 2, to ensure that the CPLs had been correctly identified. These CPL types were created to raise the abstraction level of some of the CPLs by removing or merging their characteristics and thereby make them more generalizable for other contexts. Some of the characteristics that were analyzed were,

– Origin or cause of the CPL (VGT tool IDE, the image recognition, the manual test specification, etc.).
– Where it was observed (Case 1, Case 2 or both).
– Commonality to other CPLs (Iterative analysis of above characteristics).
– Impact on transition or usage of VGT (Criticality of the CPL).
– CPL solution (Solved during the project, feasible to solve in theory, not solvable)
– Etc.

The two projects were performed mutually exclusively from one another, meaning that the researcher performing the VGT transition in Case 1 had no contact with the industrial practitioners in Case 2. All communication in Case 2 was performed by the other research team members. Thereby ensuring that corroborating evidence or information from one of the projects was not influenced by the other project.

After identification, the CPLs were categorized into four tiers, with the 58 individual CPL observations on the lowest level of abstraction, i.e. Tier 4. The Tier 4 CPLs were then divided into 26 unique CPL types, Tier 3, that were then collected into eight types on a higher level of abstraction, Tier 2 CPLs, which could be grouped even further into three top tier CPL types, 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. Additionally, the characteristics listed above were also taken into consideration.

Furthermore, this analysis revealed four high-level solutions that could be derived from the specific solutions that had been identified during the two projects. These four identified solutions are estimated to solve 50 percent of the identified CPLs and are presented in Section 4.5. The retaining 50 percent of the CPLs were either not solved or solved using ad hoc practices and solutions that were only applicable for specific CPLs. For instance, the VGT tool only supported a US keyboard which made it impossible to input special characters from the Swedish alphabet. This was solved by entering the ascii codes for said letters instead. Thus, solving the problem, but it could not solve any other CPL and it was only used in Case 1.

Project cost and return on investment was also evaluated during the study by collecting quantitative metrics from both cases. In Case 1 the metrics were recorded in detail during the VGT transition. However, in Case 2 they were obtained primarily post project completion, i.e. during the interviews in Phase 3 of the study. In order to mitigate estimation bias the reported values were triangulated through the interview transcripts and previously collected data from Case 2. All information analysis was performed by a member of the research team that did not take part in the VGT transition project in Case 1 to mitigate bias in the results. It should be noted that due to the differences between the contexts of the two cases, many of the quantitative data points are context dependent and therefore only limited generalization can be done of said data, see Section 4.6. Finally, all conclusions drawn from the analyzed information were reviewed and validated by the study participants to eliminate bias introduced during the analysis.

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 [Börjesson and Feldt(2012)]. A thorough document analysis of the system under tests (SUT) manual test suite was also performed to evaluate if the previously developed VGT test scripts were still applicable. The document analysis was necessary because the version of the SUT differed from the system that had been used in [Börjesson and Feldt(2012)]. In addition, the document analysis was required for the research team member performing the VGT transition to get familiar with the SUT and our previous work. Because the VGT transition was performed at the company, any questions that the research team member had could quickly be resolved by asking a developer at the company, consulting the SUT documentation or asking one of the other research team members. The research team member who performed the VGT transition held a post-graduate degree in computer science but had no previous industrial experience.

Furthermore, to ensure information quality the research team defined a thorough documentation process at this stage of the project. The process defined a set of quantitative and qualitative metrics and practices how these metrics should be collected for each developed VGT script and/or the VGT test suite. These metrics included, but were not limited to, development time of each test step and the entire test case, execution time of the test case, lines of code for each test step and the entire test case, CPLs found during development, 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, a GUI component, the VGT tool or the system. Practically, the data collection was done on printed forms or Excel sheets. All information was later transferred to Excel to simplify the analysis.

Even though the version of the SUT 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 [Börjesson and Feldt(2012)], were applicable. However, all of these 33, applicable, test cases could be automated and thereby constituted a complete test suite of Sikuli scripts for that particular version of the system.

As stated, the SUT was distributed over three computers, connected to one another over a local area network (LAN), which required the human tester to physically move around during manual system testing. In order to give Sikuli this capability, the tool was paired with a third part virtual network connection (VNC) software that provides support for remote computer control by streaming the target computers desktop to the viewing computer. VNC servers were added to the three computers of the system and connected, over LAN, to a fourth computer with a VNC viewer application that the Sikuli scripts were executed against.

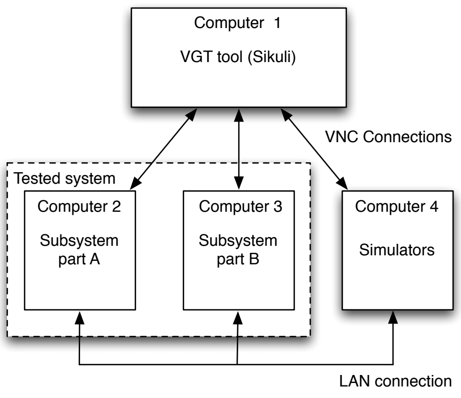

Fig. 2 The setup of the tested system, including all computers that were connected in a local area network (LAN), accessed by Sikuli using VNC.

Consequently, the Sikuli scripts were executed remotely. In the continuation of this manuscript these four computers will be referred to as the test system which has been visualized in Figure 2.

Once the test system had been setup, an architecture was developed for the VGT test suite. The VGT scripts were implemented one at a time using the manual test cases as specifications. Thus, ensuring that each VGT script was a 1-to-1 mapped representation of the manual test case with equivalent GUI interactions and assertions. This VGT transition was possible because the test steps in the manual test cases were defined in sequential scenarios. The VGT suite architecture and script development will be presented in detail in Section 3.4 33 manual test case descriptions where transitioned into VGT during the study, resulting in 33 equivalent automated VGT scripts. Several of these manual test scenarios were however applied to all components of the GUI, e.g. the same test scenario was applied on several different buttons to verify their functional conformance to their specification. Thus making the actual number of tests cases greater than the number of test case descriptions.

During the VGT transition, the test scripts were executed after each test step had been developed and verified against the results of corresponding manual test step. The automated test steps were implemented as individual Python methods that were called in sequential order to perform the test scenario. This implementation approach made the scripts modular and allowed test steps to be reused. Modularity was one of the keywords for the VGT transition to ensure reusability and maintainability of the scripts. Modularization made it possible to run the Python methods out of order, thereby shortening script verification cost, since the entire test script did not have to be re-executed every time new functionality had been added or changed. Maintainability was also achieved since the Python methods are interchangeable and can simply be replaced if the scenario or functionality of the system under test is changed.

A second keyword for the VGT transition was robustness. Robustness was achieved by implementing the scripts with three levels of failure redundancy for critical functions. The redundancy was put in place to mitigate catastrophic failure either due to image recognition failure, script failure, test system failure or defect detection in the tested system. This redundancy mainly consisted of several levels of exception handling that was implemented using Python.

As a final step of the VGT transition process, 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 script(s) were corrected and validated during execution of the entire VGT suite, i.e. execution of all the test cases in sequence. The validation process was rigorous and performed by comparing the outcome of each automated test step with the outcome of corresponding manual test step. This process was time consuming but required to ensure VGT script quality.

3.3 Detailed data collection in Case 2

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 [Yeh et al(2009)Yeh, Chang, and Miller], eggPlant [TestPlant(2013)] and Squish [froglogic(2013)], to identify the most suitable tool to fulfill the company's needs[3]. After the evaluation, Sikuli [Yeh et al(2009)Yeh, Chang, and Miller] was identified as the most suitable alternative for the transition. Further information regarding the VGT tool selection can be found in [Alegroth et al(2013a)Alegroth, Feldt, and Olsson].

The industrial practitioners held postgraduate degrees and had several years, i.e. 1-3 years per person, of industrial experience in testing among them. The majority of the VGT transition was performed by two testers. However, towards the end of the project a third tester joined the team but this person's contributions to the project was considered negligible.

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 a manual test case. The deviations from the 1-to-1 mappings were required since the manual tests, defined as use cases, were not mutually exclusive from one another. This lack of mutual exclusion originated from the depth of GUI, i.e. its many views, which for some tests required certain use cases to be executed in order to put the system in a specific state for the assertion in a subsequent use case. Therefore some of the use cases that were used often were grouped together into a single script that could be called to setup the system in a given state before continuing the execution with more independent use cases. However, the manual test case architecture in Case 2 was perceived to provide the VGT suite with a higher degree of flexibility and reusability than the manual test architecture used in Case 1. The reason was because the test cases consisted of small interchangeable pieces, i.e. the use cases, which made the creation of new test cases very quick and simple. The manual test suites that were automated consisted of several hundred individual use cases that were transitioned into an estimated 300 automated test cases. An exact number cannot be given because of the flexible coupling between use cases that made it possible to create many test cases from just a few scripts.

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. The order of test case execution could also be saved in the script. Thus, very little effort was required to setup and run the test suite.

To acquire information during the study, an information acquisition process was proposed to the industrial practitioners for the collection of quantitative and qualitative metrics. However, because Case 2 was performed during an actual development project in industry, the process could not be followed consistently due to time and cost constraints. Thus, lowering the quality of the collected information. This lack of information quality was compensated with triangulation and the interviews that were held in relation to the second workshop. The information collected by the industrial practitioners was conveyed in a continuous manner to the research team, as shown in Figure 1, through e-mail and over telephone.

3.4 The VGT suites

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

The input for the VGT transition were the scenario-based test case specifications that were used for manual system testing in Case 1 and 2. In Case 1 the test specification constituted a complete test suite with 67 manual test cases that were defined in table form. Each table constituted a test case scenario where each row was a test step of the scenario. The columns in turn defined the input and expected output. A test step was considered successful if the observed output was equal to the expected output. Further, if all test steps were successful in a test scenario then the test case was successful. The minimum number of steps (rows) in a test case was 1, the average was 7.88, median 6 and the maximum was 22. However, it should be noted that the number of manual test steps was not indicative of the effort required to run the scripts manually or to automate the tests since the level of detail in the natural language descriptions often differed. As such, a script with only a few steps could take longer to automate than a test case with many steps because these few steps defined many interactions with the SUT, whilst scripts with many steps often defined one interaction per step. These deviations originated from the test scripts being written by different people at different points in time. The tables were defined in a 126 page long Word document.

In Case 2 the test specification consisted of small use cases. Small in this context means only one or two human interactions with the system per use case. The use cases were defined as a custom sequence diagram notation with a bubble(s) showing what input the tester should give to the system on the left,

[3] eggPlant and Squish are commercial tools developed by TestPlant and Froglogic respectively.

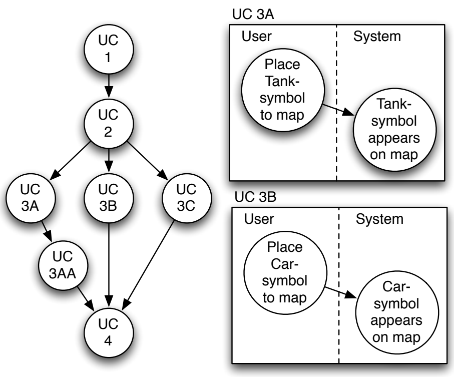

Fig. 3 Conceptual example of a test case scenario design, used in Case 2, based on loosely linked use cases (to the right). In the example the test scenario (to the left) contains three unique test-paths, i.e. test cases, that were, prior to the VGT transition, executed manually. **UC** - Use case.

and system's expected response in a bubble(s) to the right. However, these uses cases only defined the test steps. These use cases, examples shown in the right of Figure 3, were then tied together on a meta-level to form test scenarios, as shown in the left of Figure 3. Figure 3, 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 3). However, the manual test structure in Case 2 also allowed test scenarios to be of unequal length, exemplified with UC 3AA, bottom left of Figure 3. The three different test chains are then joined again, and completed, by UC 4 (Bottom of Figure 3). This architecture allowed the testers to switch use cases within the graph and thereby create different test scenarios. A test scenario was successful if all expected outputs, from all uses cases in said scenario, were observed. Each interaction or observed result, i.e. bubble in the sequences, was defined in one to two sentences of natural language.

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 component tests and includes special assertion methods for this type of testing, referred to in the tool as "unit tests" on GUI level. However, Sikuli does not have support for creating test suites of several individual unit tests. As such, to develop an entire test suite using Sikuli's standard functionality, all test cases have to be grouped into one file. However, this has negative effects on the reusability, maintainability, usability, etc., of the tests.

Custom test suite solutions were therefore developed in both Case 1 and Case 2 by using Python's object orientation and its ability to import scripts into other scripts. Both VGT suites consisted of a main script that imported the individual test cases and executed these according to an order specified in a list in the main script. This list, or lists, could easily be defined manually or be generated at runtime to, for instance, randomize the order of the test case execution. Thus, the individual test cases were coupled using the main script. However, neither data-driven nor keyword-driven testing was used even though the tests could be refactored to support these properties. The general architecture for the two VGT suites is visualized in Figure 4. 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 followed the same framework as typically used for automated unit tests, e.g. JUnit [Cheon and Leavens(2006)]. This pattern was supported by creating a skeleton script that was reused as the basis for all test cases.

In addition, 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. Each test step was also defined as an individual method to increase modularity. The modular, and hierarchal, architecture helped shorten development time by simplifying reuse, but also improved maintainability of the scripts since all components were easily accessible. Modularization also improved robustness since it became easy to encapsulate all method calls to test steps with exception handling. Thus, if a test step failed, the exception could be handled both locally in the individual scripts or on a global level in the main script.

All assertions were conducted through visual comparison between expected and actual output from the SUT's GUI using Sikuli's image recognition capability. This was achieved using branch statements that were defined as: *if expected graphical output, then report pass and continue to the next test step, else report fail and rerun the assertion.* The reason why the assertion was rerun if it failed was to verify that

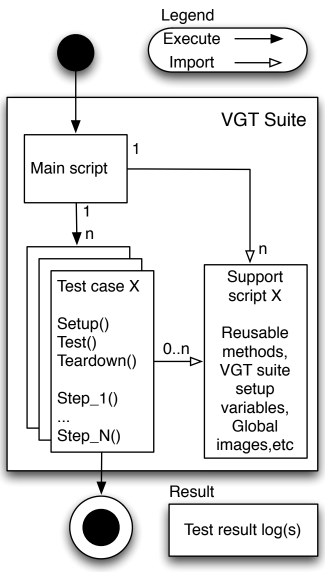

Fig. 4 The architecture of the VGT suites developed in Case 1 and Case 2. Each test script included a setup, a test and a teardown method. The test method in turn called the individual test step methods, thus separating the test step execution order from the actual test step implementation. This design perceivably improved maintainability and reusability of the test cases.

the cause of the failure was a fault in the SUT rather than a false positive caused by, for instance, failed image recognition. In Case 1, in addition to the above functionality, a second failed assertion resulted in the SUT being rerolled to a known state and the entire test case being rerun. If the same test step would fail again, the entire system would be restarted and the test case rerun a third time. This redundancy was implemented in order to with absolute certainty identify the cause of a failure. In Case 2 the only action that was taken after a failed assertion was to rerun the assertion.

Another 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 each test case. 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 file with similar graphical format to the output from automated unit tests, e.g. JUnit [Cheon and Leavens(2006)]. However, Case 2's VGT suite did not record video clips of failed test scenarios, instead it took screenshots of the tested system's faulty state when a test case failed. The screenshots were taken using a Java library that was called from the Python scripts.

The reason for adding screenshots and video recordings to the VGT suites' output was to provide the SUT's developers with more information to simplify defect identification and recreation. This functionality also helped the VGT script developers to distinguish defects in the SUT from defects in the VGT suite itself.

It should be noted that the VGT suite architecture presented in this manuscript is generally applicable but not necessary in most other tools than Sikuli [Yeh et al(2009)Yeh, Chang, and Miller]. The reason is because most VGT tools, e.g. JAutomate [Alegroth et al(2013b)Alegroth, Nass, and Olsson] and EggPlant [TestPlant(2013)], have built in support for test suite creation and maintenance. As such, these tools also have built in failure mitigation and exception handling support. However, for individual test cases, we propose a general practice of using the setup, test and teardown design as described by unit test frameworks. This pattern encourages modularity and effectively separates the tests preconditions, tests and postconditions.

As stated, the test cases were implemented as 1-to-1 mappings of the manual test cases using a structured process where each test step was implemented and verified against the manual test case outcome. Furthermore, if a test script failed during test execution the failure would be evaluated to verify that it was caused by a SUT defect, i.e. evaluation of the assertion to verify its outcome. In Case 1 this practice was considered sufficient for verification since the system did not evolve during the course of the project. However, since the VGT suite was being developed in a real project in Case 2, all test cases and assertions were verified with even intervals through complete executions of the manual test suite. This practice verified that no test cases that should have failed were reported as successful. However, due to cost constraints, only the standard test case execution order from test case 1 to n was evaluated. Thus, it is possible, but unlikely, that there could have been assertions that failed to report defective SUT behavior. However, it is unlikely that assertions that should have failed instead succeeded because of the sensitivity of the image recognition that is more prone to failure than to succeeding when the expected output is not present on the screen. However, since this behavior cannot be ruled out, a theoretical mitigation practice is to also verify succeeding test scripts randomly post development.

4 Results and Analysis

This section will present the main contributions of this work including the Challenges, Problems and Limitations (CPLs) related to the transition to and usage of VGT, potential solutions to these CPLs and results regarding the bug-finding ability and ROI of the VGT transition. The Tier 3 CPLs (The 26 CPL types) are presented in the following subsections according to the order of the Tier 2 CPL types as they are presented in Figure 5. In addition, each Tier 3 CPL has been marked in the text with the number proceeding its name in Figure 5, i.e. CPL type x is marked as (x) in the text. Furthermore, in order to provide the reader with an overview of the distribution of the 58 CPLs (Tier 4), each subsection is proceeded with a table how many CPLs there were of each type.

Since the number of CPLs and types of CPLs are extensive we have summarized the main results for each type and provide a summary in Section 4.4, including a table listing the most prominent CPLs, Table 5. The summary section also analyses if the identified CPLs have been found, or are likely to hold, also for alternative test techniques. Table 6 summarizes the outcome of this analysis.

The result section then continues with a description of the identified solutions to CPLs in Section 4.5. These solutions have also been summarized in Table 7. This is followed by results regarding the bug finding ability of the VGT test suites and quantitative results on the development time, execution time, etc., and the evaluated return on investment of the VGT transition, summarized in Table 8. These results are presented together with a discussion regarding the impact of the results for industry and a theoretical model for the ROI in a general VGT transition project, presented in Figure 8.

4.1 Test system version CPLs (Tier 1)

The Tier 1 CPL type related to the tested system (SUT) was split into five Tier 2 CPL types, as shown in Figure 5. These Tier 2 CPL types are related to the version of the SUT, the general SUT, defects in the SUT, the company or the SUT's simulators. The following subsections will describe the CPL types related to these categories.

4.1.1 Test system version (Tier 2)

Complex systems with several versions and/or variants are associated with test related CPLs that affect all types of testing, including VGT. For instance, these system's require appropriate version control to ensure alignment between code and system specifications, e.g. test specifications. This CPL was observed in Case 1 where the SUT was a demo build of the product, intended for customer demonstrations rather than customer delivery. As such, the SUT did not include all the functionality of the product (1), had a faulty test specification (2), etc. The SUT was used during the project because of resource constraints (14), i.e. whilst the product SUT required 12 computers, including three servers, the demo system required only three computers in total. Thus, lowering the costs for setting up the SUT and the VGT transition project overall. Resource/budget constraints is a common CPL in implementation and improvement projects in the industrial context which limits the amount of evaluation and implementation that can be done.

The lack of documentation for the SUT in Case 1 required the research team to use a test specification for another version of the system which was not aligned with the SUT's functionality (2, 13). In addition, the research team member who performed the VGT transition in Case 1 lacked domain knowledge which made it difficult to evaluate which manual test cases were applicable before the starting a new script. As such, several scripts were already partially implemented before it was identified that they could not be completed. Thus, adding unnecessary costs to the transition project. Note that the cause of this CPL was the lack of SUT functionality (1) rather than limitations in the VGT tool.

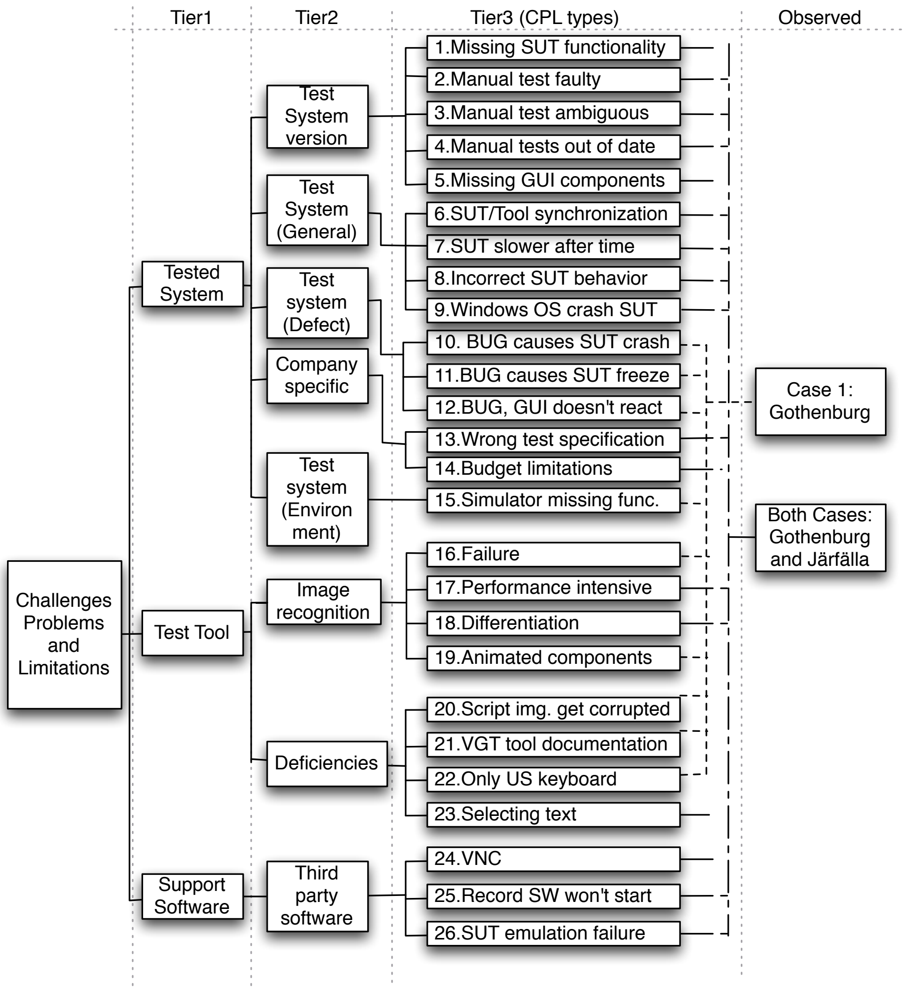

Fig. 5 A hierarchal tree diagram over the Challenges, Problems and Limitations (CPLs), Tiers 1-3, that were identified in Case 1 and Case 2. **SUT** - System under test, **VNC** - Virtual Network Connection, **GUI** - Graphical User Interface, **OS** - Operating system, **SW** - Software, **Func.** - Functionality.

In addition, several of the manual test cases tested the SUT's functionality incorrectly (2) or were out of date (4) and no longer valid because of changes to the SUT. Other tests were ambiguous (3) or aimed at testing functionality that had been removed (2). These test cases were reported to the company and were considered a positive side effect of the VGT transition. Thus, this CPL relates to the complexity of keeping test specifications up to date (4) as a mature system evolves into versions/variants over time. Consequently, it is important to evaluate the manual test suite before VGT transition to ensure that the test cases are applicable. Failure to do so will result in additional cost and erroneous automated test cases that in the worst case can lead to SUT defects propagating post customer delivery.

However, identifying if a manual test specification is faulty requires domain and SUT knowledge, which, as previously stated, the research team member performing the VGT transition lacked. This lack of domain knowledge resulted in additional cost to the project because the company's developers had to be consulted to, for instance, interpret ambiguous manual test cases, explain SUT functionality, etc. As such, it is perceived that VGT transition will be more effective if it is performed by a domain and/or SUT expert. A statement that is corroborated by the practitioners in Case 2 who reported that they had not found any test case that they could not automate.

In our previous work [Börjesson(2010)] we automated manual test cases for a version of the system with 50 manual test cases. However, for the main branch version of the system there were 67 manual test cases, of which only 33 were applicable on the version of the SUT used in Case 1. Hence, restricting the usability and portability of the developed VGT test suite for other versions or variants of the SUT.

Lack of functionality that prohibited automation of more manual test cases were, but not limited to, the SUT version's lack of operator roles (1), missing GUI components (5), missing radar functionality (1),

CPL category	CPL sub-category	Nr. of (Tier 4) CPLs	Percentage out of the 58 CPLs
Test System	Test System version	20	34.4
	Test System (General)	6	10.3
	Test System (Defects)	6	10.3
	Test Company specific	1	1.72
	Test System (Environment)	1	1.72
	Total	37	58.44

Table 1 Summary, and distribution, of problems, challenges and limitations (CPLs) related to the tested system out of the 58 CPLs that were found in total.

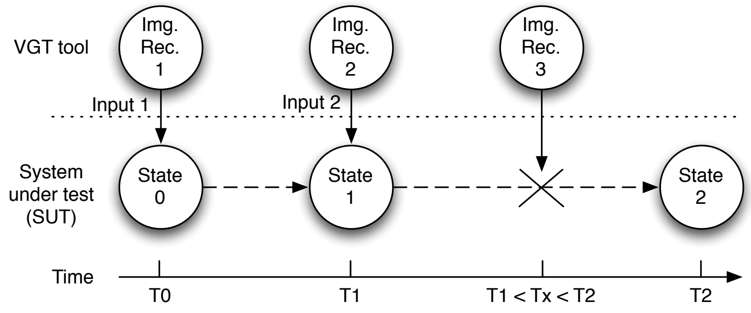

Fig. 6 Visualization of the CPL that the VGT tools' image recognition algorithms requires less time to find a match on the SUT's GUI than the time required for the SUT to transition from one state to another. **Img. Rec.** - Image recognition.

missing simulator support, etc (1). This lack of functionality also made it impossible to run the manual test cases. This missing functionality had purposely been omitted to scale down the amount of required hardware to run the SUT and make it transportable for demonstration at conferences, workshops, etc. As a consequence, the developed VGT suite could only partially test other versions or variants of the SUT. However, for the manual test cases that were applicable the study participants in both Case 1 and 2 reported that they did not find any test case operations that they could not automate using the VGT tool.

In summary,

– Misalignment between system implementation and test specifications leads to partially completed test cases, faulty test results and additional transition cost.
– Missing system functionality leads to only partially completed test cases.

4.1.2 Test system (General) (Tier 2)

One of the CPLs that was reported to cause the most frustration during the study was that the VGT script execution had to be synchronized with the SUT's execution (6). This CPL has been visualized in Figure 6. In the figure, the VGT tool's image recognition is successful at time T0 and input is sent to the SUT which triggers a transition from State 0 to State 1. At time T1 the image recognition is once again successful because it occurs after the SUT has reached a stable state, State 1. However, at time Tx the image recognition fails because the SUT is currently transitioning from State 1 to State 2 where the sought GUI component is not yet visible on the SUT's GUI. Thus causing a script failure.

This is a common scenario during VGT script development and requires the image recognition at Tx to be delayed until the SUT has reached the stable state in State 2. There are two approaches to add delays in, to the best of our knowledge, all VGT tools (6). The first approach is to delay the script dynamically until a sought GUI component is visible and the SUT is perceived to have reached a stable state. The second approach is statically wait for n number seconds before proceeding with the script execution. Dynamic delays are more robust to SUT performance change but can fail because they rely on the VGT tool's image recognition algorithm. In contrast the static delay is completely robust but slows down script execution since the static delay has to be based on worst case SUT state transition time.

However, even though delays improve VGT scripts robustness, they also add maintenance costs since they require realignment over time to match the SUT's performance (6,7). Dynamic delays also require the sought GUI component (bitmap) to be updated over time. The practitioners in Case 2 reported that adding delays was frustrating because the script had to be rerun to verify its correctness. Additionally, since the worst case state transition time could only be estimated, static delays required some trial and error before they became robust.

Synchronization was also a CPL in Case 1 because it was observed that the SUT's state transition time increased after a couple of days if the computers were not restarted (7)[4]. Similar observations were made in Case 2, causing test scripts that had previously succeeded to fail (7). The solution for this CPL was simply to restart the computer(s) and it is therefore considered a minor CPL. However, the CPL is still worth mentioning since dips in state transition time shorter than a human can observe can cause the scripts to fail and cause frustration among the test script developers.

It should be noted that the synchronization CPL between scripts and the SUT is common to other automated testing techniques and that the general solution is to add delays to the scripts, as presented above (6).

Another CPL concerns unexpected SUT behavior (8) caused by events that are not part of the test scenario but which can interrupt the test script execution. This type of behavior can be caused by the SUT, the operating system (9) or third party software used to run the tested system or its surrounding environment. Events which cause this type of behavior are popups, e.g. software update messages or error messages, which are launched by the operating system, sudden performance dips due to, for instance, a new process being started by the operating system, hardware failure or a defect in the tested system, etc (8, 9).

In summary,

– Synchronizing script execution with system execution is time consuming and one of the main sources of frustration during VGT transition.
– Unexpected system behavior triggered by the system under test, operating system, etc., causes a lot of frustration and requires additional failure mitigation code.

4.1.3 Test system (Defect) (Tier 2)

During execution of a VGT suite practice, i.e. during regression testing, a VGT script would terminate after identifying a defect, report the defect in the test output, 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 only be automated up to step n since the defect would prohibit any interaction with the tested system after this step. Additionally, even if the execution would not have been prohibited, all execution after identifying a defect at a test step could be in an invalid and/or useless system state that would not appear in real-world use of the system. Furthermore, continued execution after a test step has failed has a higher probability of reporting a false positive test results. Hence, reporting both test step x and test step x+n, dependent on the interaction at step x, as failed even though step x+n had passed if step x had passed. However, since the failure at step x could have been caused by failed image recognition it should not be assumed that the first failure shows the defect and the second is a false positive. Thus, both failures need to be investigated to find their cause, which adds cost. Therefore, the test suites in both Case 1 and 2 stopped script execution after an assertion failed, rolled back the system to a known state and continued with the next test case.

However, not all SUT defects that lead to SUT failure are recoverable, many cause the SUT or the SUT's GUI to freeze or crash (10, 11, 12). A human tester can resolve this by restarting the application, but for the VGT scripts it poses a CPL. If the defect in the SUT causes the SUT's computer to crash or freeze then the VGT scripts can no longer access the SUT or resolve the situation (10, 11). However, if it is only the SUT that crashes or freezes, exception handling can be built into the VGT suite that automatically restarts the SUT.

Exception handling of this type was implemented in Case 1 where a crash would result in the VGT application restarting the operating system of the SUT's computer. This was possible since the VGT suite was run remotely over a Virtual Network Connection (VNC).

Six SUT defects were identified during the study, of which four were previously known to the company in Case 1. However, the study also revealed two defects that were previously unknown or only partially known to the company. The reason why these previously unknown faults were uncovered by VGT was because of the quicker and more cost effective execution of the VGT suite compared to manual tests. This VGT property made it feasible to rerun test cases to find faults that only appeared sporadicly during SUT interaction. One such defect, observed in Case 1, was in a tab menu in the SUT that did not always load

[4] This behavior was only observed on the project version of the SUT.

properly during system startup. In combination with the video recording functionality of the VGT suite, described in Section 3.4, the faulty behavior of the system could be determined and reported in detail to the company. This example shows how the video recordings adds to defect finding ability. The identified defects were of different type, from defective GUI functionality, e.g. tabs not launching, to complex defects 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. All of the defects were considered critical because of the safety-criticality of the SUT.

In Case 2, three unknown defects were identified in the SUT in addition to known defects found by the previously used manual testing. However, the reason why the defects had previously not been identified was different from Case 1. The manual test cases in Case 2 were always executed in order from test case 1 to test case n, once every development iteration, i.e. every six months to one year. Furthermore, due the company's strict testing budget, not all of the manual test cases could be executed each iteration. In contrast to the manual tests, the VGT scripts could be executed several times every week and several times in row each time with different orderings among the test cases. These VGT properties lead to the discovery of the previously unknown defects, which the practitioners in Case 2 reported would not have been found if it hadn't been for the VGT transition.

The defect finding ability of VGT is a strength but also presents a possible CPL, i.e. that a company becomes so confident with VGT that it is used as a substitute for manual testing. However, VGT is not perceived to be a replacement for manual testing, primarily because the technique can only find defects that are covered by an assertion in the test scenarios [Berner et al(2005)Berner, Weber, and Keller]. In contrast, human testers can also identify defects that are triggered by a scenario but not covered by an assertion. Hence, manual scenario based and exploratory testing is still required to complement VGT to uncover defects [Itkonen and Rautiainen(2005b)]. A statement that was corroborated by the practitioners in Case 2.

The results regarding how the unknown defects were identified also shows the importance of test case execution order. Executing test case x after test case y may not have the same outcome as executing test case y after test case x. In addition, executing test case x twice in a row can also lead to another outcome. Thus, providing support that the low cost of running a VGT suite can help uncover new defects by supporting multiple test executions in a row with different predetermined, or even random, test case orderings. A statement corroborated by [Rothermel et al(2001)Rothermel, Untch, Chu, and Harrold] since different orderings cover more system states and sequences of interaction that may appear during real-world usage of the system. Completely random ordering can result in test case execution scenarios that may never appear in practical use but still add value by exposing what features of the SUT are associated with faulty system behavior. However, how to find the optimal, or even most suitable orderings that identify defects is still a CPL. During the study, the orderings were only evaluated randomly, i.e. no systematic practice was found to improve the defect finding ability of a VGT suite.

A prerequisite for a VGT suite to be used effectively with different test case orderings is that the test cases have been developed in a consistent and suitable manner that make them interchangeable within the test suite. As such, the test cases and overall test suite should adhere to a well defined architecture or framework. For Sikuli test suite developers this presents a CPL that is more related to traditional software development than testing since Sikuli does not have built in support for test suite creation. Consequently, efficient test suite development is a common problem for all VGT tools but more explicit in Sikuli.

In summary,

- Defects in the system can cause it to crash or freeze, which requires failure handling code in the VGT scripts.
- More defects can be found with VGT since there is no cost associated with running the test suites which allows it to be run with different test case orderings. However, finding what orderings identify more/less SUT defects is still an unexplored CPL.
- A good test script architecture is essential for test suite usability.

4.1.4 Company specific (Tier 2)

The manual test specification used in Case 1 was not aligned with the SUT's implementation (13) as explained in Section 4.1.1. The research team member in Case 1 also lacked the domain knowledge to identify this misalignment which later required several partially completed test cases to be refactored. Thus, raising the cost of the VGT transition. However, the source of this CPL is general to all testing and relates to the complexity of product management [Ebert(2007)]. Product management that becomes increasingly more difficult as the number of artifacts, e.g. test specifications, increase for different versions

CPL category	**CPL sub-category**	**Nr. of (Tier 4) CPLs**	**Percentage out of the 58 CPLs**
Test Tool	Test tool (Sikuli)	13	22.4
	Test scripts	1	1.72
	Total	14	24.12

Table 2 Summary, and distribution, of problems, challenges and limitations (CPLs) related to the test tool (Sikuli) out of the 58 CPLs that were found in total.

and variants of a system. Hence, this CPL indicates that any VGT transition project should be driven by a domain and/or test system expert in order to mitigate unnecessary development cost.

Another company specific CPL in Case 2 was the VGT transition project's limited budget (14). 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 general to all types of process improvement.

In summary,

– Improper version control and budget constrains set by the company affects the VGT transition successfulness, test case quality and test case results.

4.1.5 Test system (Environment) (Tier 2)

The SUT in Case 1 had several external interfaces to control an airport's landing lights and radar during operation. Since these external systems, e.g. the radar, are not available during testing the company instead uses simulators to stimulate the SUT's external interfaces. Software simulation of hardware is a common practice in industry [Mongrédien et al(2006)Mongrédien, Lachapelle, and Cannon, Börjesson and Feldt(2012)]. However, simulators are often SUT version/variant dependent (15). As such, misalignment between the SUT and the used simulator will limit the SUT's functionality (1) and in continuation what test cases can be applied. Thus presenting a version control CPL that must be taken into account during VGT transition and execution.

In summary,

– Improper simulator support for the system under test leads to incomplete test cases and test cases that will require more maintenance.

4.2 Test tool (Tier 1)

In the following section we will focus on CPLs related to the VGT tool. Some of these CPLs are Sikuli specific, whilst others are perceived as general to other VGT tools as well. Two sub-categories of CPLs were identified as VGT tool specific. First, CPLs related to the tool's image recognition. Second, CPLs related to deficiencies in the VGT tool such as lack of documentation, bugs in the IDE, etc.

4.2.1 Image recognition (Tier 2)

Sikuli is a tool developed and maintained in an open source project that was started at the User Interface Design Group at the Massachusetts 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 (version X-1.0rc3), i.e. not a finished product. Therefore, the tool contained software defects that made the tool's integrated development environment (IDE) and image recognition unstable.

Sikuli's image recognition algorithm instability caused it to randomly fail in cases where it had previously succeeded, click on generic positions next to the sought images, etc. These failures were observed in both Case 1 and Case 2 and were another source of frustration during the VGT transition because of the random nature of the CPLs and because no solution could be found to mitigate them. The source of the CPL is perceived to be software defects in Sikuli (16).

Another cause of image recognition failure is the image recognition algorithm itself because it is performance intensive (17). The performance requirements can cause the SUT to run slower than in normal use, which can lead to faulty SUT behavior and perceivably random image recognition failure.

However, not all image recognition failures identified in the study were random. Sikuli uses a fuzzy image recognition algorithm that allows the user to set the required similarity level for the sought image to be a match. If the similarity is set too high or too low it is possible that the tool will find an incorrect match or no match at all (18). Fine-tuning the image similarity to assure correct image identification was together with the synchronization CPL, presented in Section 4.1.2, reported to be the two main contributors to development cost and frustration during the VGT transition projects.

To help the script developer to set the image similarity, 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 screen, indicated with a square. If several matches are found, these are ranked using colors ranging from blue for a poor match to red for a good match. During script execution, Sikuli will always interact with the image with the highest similarity, if not told otherwise. According to the study participants, corroborated by the authors' own experiences, the preview feature has never failed to find a match. Thus, indicating that the image recognition algorithm has a 100 percent success rate. However, it was reported from both Case 1 and 2 that even though the preview feature finds a match, this does not guarantee that the script will succeed during execution (18). Thus, the preview feature's behavior does not always match the script execution behavior, leading to confusion and frustration among the script developers.

Another image recognition CPL concerns Sikuli's inconsistent ability to find certain types of bitmap objects, for instance animated images (19) and images with similar appearance but with slightly different color (18). This functionality was required in Case 1 since the SUT had animated buttons that indicated alarms to warn the user. A conclusion was therefore drawn that Sikuli is not equally usable for animated GUIs (19). This conclusion is corroborated by results from our previous work [Börjesson and Feldt(2012)]. However, the SUTs in Case 1 and 2 were mostly non-animated and therefore this CPL did not prohibit the use of VGT. In our previous work [Börjesson and Feldt(2012)] we also showed that another VGT tool, CommercialToll, was equally susceptible to this CPL.

Secondly, it was possible to switch the color set of the SUTs GUI in Case 1, from light gray to dark gray, to make it more comfortable in different lighting conditions. However, this presented a CPL for Sikuli that could not always distinguish between the lighter and darker colors and therefore this SUT functionality could not be tested with complete confidence. 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. Thus, this CPL provides support for the need to also do complementary manual testing.

In summary,

- Sikuli's image recognition algorithm sometimes fails even though the image recognition preview feature finds a match, which added a lot of frustration to the VGT transition.
- Sikuli's image recognition has lowered success rate when attempting to recognize images that are very similar or that are animated.

4.2.2 Deficiencies (Tier 2)

Sikuli script is a Python-based scripting language with unique methods that incorporate the tool's image recognition capabilities. However, there is no comprehensive API documentation for the language, which added confusion and frustration during the study because not all of the language methods are intuitive (21). One such method is called wait, which is a delay method that takes an image and/or a delay in seconds as input. However, wait(image, delay) and wait(delay, image) have different properties. Whilst the first alternative waits for a maximum of delay seconds for an image to appear, the second always waits delay number of seconds and then tries to find the image. However, as reported in our previous work [Börjesson and Feldt(2012)], Sikuli script is in general an intuitive scripting language even for novice users.

The transition approach used during the study was to implement all VGT scripts as 1-to-1 mappings to the manual test cases. This approach has several benefits, including improved verifiability of the scripts because they can simply be compared to the manual test execution. However, this approach is not always applicable and not always the most efficient. As reported in Section 3.3 it was not possible to implement all test cases in a 1-to-1 fashion in Case 2, because of dependencies among the use cases that formed the test scenarios. The practitioners in Case 2 therefore stated that future development would not be based on the manual test cases but rather on the testers' domain knowledge and that the test cases would first be developed for the core functionality of the SUT. Thus, the SUT features that were the most stable and were the most essential from a business perspective. However, this presents a CPL regarding script complexity and how to verify script correctness.

Another CPL that must be taken under consideration during VGT transition is that VGT tools, even though they emulate human user behavior, are not human. Therefore, the tools sometimes perform GUI

CPL category	**CPL sub-category**	**Nr. of (Tier 4) CPLs**	**Percentage out of the 58 CPLs**
Support software	Third party software	10	17.2
	Total	10	17.2

Table 3 Summary, and distribution, of problems, challenges and limitations (CPLs) related to the support software, e.g. VNC, out of the 58 CPLs that were found in total.

interactions in unnatural ways that can cause unintuitive CPLs. For instance, Sikuli can perform interactions much quicker than a human user, which can lead to faulty script behavior (23).

As stated, there are many CPLs related to Sikuli's IDE, scripts and image recognition. Through experimentation with approximately 50 computers we came to the conclusion that the stability of Sikuli is related to what version of the Java runtime environment (JRE) is installed on the computer. Hence, the tool was perceived considerably more stable on some JRE versions than other. However, no explicit information was acquired what versions improved stability the most. The effect of this CPL is that users with less advanced computer knowledge can be discouraged from using or even test the tool, if they can get it to work at all. This problem threatens the trustworthiness of VGT as a technique and is perceived to only be solvable through continued development of the SIkuli tool.

In addition, several other low-level, technical, CPLs were identified, for instance that Sikuli cannot type or read swedish letters using its optical character recognition (OCR) algorithm, frequent IDE crashes, etc. However, due to the technical nature of these CPLs, we will not go into more detail about them here. It was reported by the study participants that most of these CPLs could be solved, e.g. by running the scripts from the command prompt rather than through Sikuli's IDE, but that the CPLs caused confusion and frustration during the project.

In summary,

– Sikuli lacks proper API documentation, which added confusion during the VGT transition since some methods have several different behavior depending on the order of the input parameters, etc.
– 1-to-1 mapping between manual test cases and VGT scripts improves verifiability and maintainability of the scripts.
– Sikuli can emulate human behavior but it is not human, which has to be taken into consideration to avoid faults, e.g. Sikuli double clicks much faster than a human.
– The version of Sikuli (version X-1.0rc3) used in the study was sensitive to what Java runtime environment was installed on the computer running tool.

4.3 Support software (Tier 1) and Third party software (Tier 2)

Virtual network connection (VNC) software was used in both Case 1 and 2 to make it possible for Sikuli to interact with the SUTs, which were distributed across several physical computers. Secondly to make Sikuli's script execution non-intrusive, which meant removing the impact of running the performance intensive image recognition algorithm 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 SUT, which could for instance slow down the SUT execution and cause script failure due to desynchronization of the SUT and the VGT script executions. .

However, as mentioned, the use of VNC was also the cause of several CPLs. First, the use of VNC was reported to lower the image recognition success rate from 100 to 70 percent because of network latency (24). Thus, even when the VNC application was running in an optimized state, 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 (19, 24). For instance the emergency nets in Case 1, since the lower frame-rate could cause the SUT's animated buttons to become distorted as they were toggling color. Another CPL is that the mouse cursor can not be removed from the rendered images sent from the VNC server to the viewer, which requires extra consideration in the script, e.g. to move the mouse cursor to an appropriate location after each interaction (24). This is a simple practice but adds script lines of code.

Furthermore, as observed in Case 1, the choice of VNC application is a relevant factor for the success of the script execution. The general conclusion is therefore that 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

CPL category	**CPL sub-category**	**Nr. of (Tier 4) CPLs**	**Percentage out of the 58 CPLs**
Test System	Test System version	20	34.4
Test Tool	Test tool (Sikuli)	13	22.4
Support software	Third party software	10	17.2
Test System	Test System (General)	6	10.3
Test System	Test System (Defects)	6	10.3
Test System	Company specific	1	1.72
Test System	Test System (Environment)	1	1.72
Test Tool	Test scripts	1	1.72
	Total	58	∼100 (99.8)

Table 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 CPL types. Tier 1 CPL types have been listed for each sub-category.

compatible with VGT. In addition, the developer should try to find the most qualitative product to avoid CPLs related to the VNC application freezing, showing the wrong colors, compression of the imagestream, etc.

In Case 1 a third party recording software, Camtasia, was also added to the VGT suite. The tool was used to capture recordings of the test suite execution which helped developers to recreate the defect and easier identify its cause. However, in several occasions Camtasia would not start properly during script execution which resulted in the VGT scripts failing as well (25). 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. Furthermore, switching between several different applications in one test case perceivably adds overhead and complexity. However, there is no empirical evidence to corroborate this claim, which is therefore a subject of future work.

Attempts were also made in Case 1 to migrate the VGT transition from the demo version of the SUT to the product version of the SUT by emulating it on 12 virtual machines on a single computer. However, this attempt was unsuccessful because the performance load of the entire system was too great (26). Hence, a more powerful computer had been required which was could not be acquired due to budget limitations (14).

In summary,

- Running VGT tools on Virtual Network Connection (VNC) software can improve VGT usability, e.g. for execution of distributed test cases and lowers performance strain on the computer running the system and the VGT tool.
- VNC lowers image recognition success rate because of image transfer latency.
- Adding a screen recording software, e.g. Camtasia, to the VGT suite improves the quality of the test results and makes it easier to replicate system under test faults manually.
- Adding more third party tools to the VGT solution/suite means that more aspects of the suite can fail, e.g. incorrect third party tool startup, crash, etc.

4.4 CPL Summary

58 challenges, problems and limitations (CPLs) were observed during this study at Saab in Gothenburg and Järfälla. Analysis of the CPLs made it possible to categorize them into three tiers, with the lowest tier containing 26 unique types of CPLs, as shown in Figure 5. Table 4 summarizes, numerically, how these 26 types were distributed over the Tier 2 CPL types and in turn how the Tier 2 CPLs were divided over the top three Tier 1 CPL types. The table also shows that most CPLs were related to the tested system (SUT) and the test environment (58,4 percent) rather than the VGT tool (24.1 percent). This is a significant result since SUT related CPLs are perceived to be more general to other test techniques, as shown in Table 6 that we will return to shortly.

In Table 5 the most prominent Tier 3 CPLs identified during the study, shown in Figure 5, have been compiled into six CPL themes. Prominence was evaluated based on added frustration, confusion, occurrence and severity of impact for either the transition to or usage of VGT. As can be seen in the table, the themes were observed in both studied cases and were spread among the VGT tool, the SUT and the support software. However, three out of the six themes are related to the VGT tool which together with the results

Nr.	CPL theme title (Tier 3 CPLs)	Description	Q-attr.
1	Test tool deficiencies (20-23)	The VGT tool used in the study, Sikuli, was an immature product and therefore included several defects, lacked proper API documentation, etc., resulting in frustration and confusion among the study participants. Some of the CPLs related to this theme could be solved with innovative scripting but it is perceived that only further development of the tool will solve the CPLs in this theme completely. As such this theme is considered Sikuli specific, even though other VGT tools may be subject to similar CPLs.	Robustness, Usability, Reusability, Learnability, Portability, Maintainability
2	Image recognition (16-19)	This theme of CPLs is unique to VGT but considered general to all VGT tools. Image recognition algorithms are technologically advanced but also complex and performance intensive, which is associated with several CPLs, including less than perfect success rate. These CPLs are however reported, from both Case 1 and 2, to be solvable through innovative scripting and exception handling. Thus providing current VGT tools with sufficient robustness to be industrially applicable. However, further advances in the area of image recognition are still required to further improve the industrial applicability of VGT.	Robustness, Usability
3	Negative VNC effects (24)	VNC constituted a significant part of the VGT test architectures used in Case 1 and 2 and made it possible for the VGT tool to perform test cases that required interaction with several different computers. In addition, VNC made the VGT script execution non-intrusive. However, VNC also lowered the image recognition success rate from almost 100 percent to 70 percent. As such, there is a tradeoff between tool usability for distributed systems and script robustness. This CPL theme is considered generic for VGT tools with the exception of EggPlant that has built in, robust, VNC support. It is therefore perceived that this theme could be solved through technical innovation from the VGT tool providers.	Robustness, Usability, Portability
4	Test specification (2-4,13)	This CPL theme is considered general to all test techniques, including manual testing, since faulty, ambiguous or out of date test cases will lead to faulty test results regardless of how the tests are performed. The CPL is caused by the complexity of maintaining a SUT over time, including version control, documentation and SUT alignment, etc. The solution for this CPL theme was outside the scope of this study but it is perceived that it can be solved through improved industrial documentation practices and tooling.	Usability, Reusability, Learnability, Portability, Maintainability
5	SUT testability deficiencies (1,5,15)	Missing SUT, GUI and simulator functionality constitute a CPL for any test technique, including manual testing. For VGT the main effects, observed during the study, was that it leads to unnecessary costs and incomplete VGT scripts. This theme is linked to theme 4 since faulty SUT behavior can be the result of improper or incomplete specifications. No solution was identified for this theme during the study but it is perceived that it would be mitigated as the functionality of the SUT increases over time.	Robustness, Usability, Reusability, Portability
6	Script tuning (6, 7, 18)	This theme was considered the most time consuming and frustrating during the study and includes VGT script/SUT synchronization and image similarity tuning. Both these practices require trial and error because no structured approach could be identified how to systematically delay or set the image similarity in a VGT script. However, reports from the study participants indicates that good estimations of where to delay or adjust image similarity can be achieved once the script developers become more experienced with the technique. This CPL theme is considered general to all automated test techniques. A statement which is corroborated by academic literature, e.g. [Horowitz and Singhera(1993)].	Robustness, Usability, Reusability, Portability, Maintainability

Table 5 Compilation of the six most prominent themes of challenges, problems and limitations (CPLs) observed in both Case 1 and Case2 during the study. The numbers in parentheses after the CPL theme titles correspond to the Tier 3 CPLs in Figure 5. These CPLs were chosen based on occurrence, perceived negative impact on the transition to, or usage of, VGT, added frustration, etc. Deficiencies are in this context attributes that cause CPLs, or allow the CPLs to emerge. **VNC** - Virtual Network Connection, **SUT** - System under test

from Table 4 indicates that even though most CPLs were SUT related, the most influential CPLs were related to the VGT tool. The table also shows that all of the six themes affect VGT's usability and most affect its robustness, which is a concern for the techniques industrial applicability. However, this observation is countered by reports from the study participants that stated that despite the technique's many CPLs, it was still valuable, cost-effective and had equal or even better fault finding ability than manual tests.

Table 6, in turn, shows the results of an indicative analysis of the CPL themes generalizability to other GUI-based test techniques and manual testing. The input for this analysis were the observations from the study and reports from academic literature, presented in Section 2. As such, no empirical evaluation was performed between VGT and other GUI-based techniques and we therefore stress that there might exist tools and practices that are exceptions to the presented results. However, the table provides an overview of the similarities between VGT and other techniques and more importantly, but not surprisingly, that the CPLs that are unique to VGT are mostly image recognition related. All other CPLs, except two, are common to GUI-based testing and manual testing, e.g. test synchronization to the SUT [Horowitz and Singhera(1993)] and usage of faulty manual test specifications that lead to faulty test results regardless of how they are executed. This result is a significant since other GUI-based testing techniques are used successfully in industrial practice, which would corroborate our previous research results that VGT is industrially applicable [Börjesson and Feldt(2012), Alegroth et al(2013a)Alegroth, Feldt, and Olsson]. However, other GUI-based techniques have been reported to be associated with high maintenance costs [Sjösten-Andersson and Pareto(2006)], costs that are still unknown for VGT. As such, further research is required to identify these costs and the impact of the VGT unique CPLs on the technique's long-term industrial applicability.

4.5 Potential CPL solutions

Based on the acquired information from the study it was possible to derive four general solutions that solve more than half of the presented CPLs. These solutions have been summarized in Table 7 and have been linked to the CPL themes from Table 5 that they solve or mitigate. The reason why precise solutions are not presented for all CPLs is because some CPLs were never solved, other CPL solutions required too much effort to be solved during the study but most solutions were ad hoc and could therefore not be generalized. Thus, in an attempt to keep the length of this manuscript down we have chosen to only present the most general solutions we identified.

The first general solution was to ensure that the scripts were developed with several levels of exception handling to mitigate CPLs such as tool and image recognition failure, failure due to SUT fault identification, etc. This solution is taken from traditional programming and can be achieved in Sikuli by using Python's exception handling. In combination with the VGT suite architecture presented in Section 3.4 it is possible to achieve a good level of script robustness. However, as reported by the study participants, making the entire VGT suite robust can be quite time consuming, especially since many exceptions are caused by unexpected tool or system behavior.

There are two practical ways of achieving robustness in Sikuli scripts. The first is to add exception handling locally in the scripts after a failure event has been observed. An example from Case 1 was in scripts that required the system to be restarted, which often caused Window's "failure to terminate process" messages to appear and deny the script from performing the next script operation. By adding a check if the message was visible after initiating the system restart, and close the message, this CPL was resolved.

The second way is to create a global solution that is triggered regardless of which exception is thrown. Hence, if an interaction with the SUT fails, the exception is sent up the VGT suite architecture to the main script that runs through a series possible mitigation operations. Some examples of these operations could be to search for buttons labeled "OK, Okey, Close, Continue" or specific buttons, such as the Windows close button found in the top right of every application window, etc. Another, more drastic global solution is to reset the system after each failure by restarting it to ensure that the SUT is in a known and fault free state. The VGT suites developed during the study were implemented with a combination of both local and global solutions.

Exception handling is perceived to be a general solution for all VGT tools that have similar scripting support as Sikuli. However, other tools, e.g. JAutomate [Alegroth et al(2013b)Alegroth, Nass, and Olsson] and EggPlant [TestPlant(2013)], also have built in failure mitigation support such as multiple image recognition algorithms, etc. Failure mitigation that relieves some, but not all, of the need for custom robustness improvements and exception handling solutions.

Sikuli lacks documentation for Sikuli script, which was regarded as a very frustrating CPL during the study. The solution to this CPL is simply to add comments to the scripts during development, e.g.

CPL theme Nr.	Tier 3 Nr.	Name	Effect	GUI-based test techniques	Manual GUI-based testing
1	20	Script image gets corrupted	Leads to required script maintenance.	N/A	N/A
	21	VGT tool documentation	Lack of documentation adds frustration to VGT transition and lowers learnability of VGT	Yes	N/A
	22	Only US keyboard	Leads to script developer/ tester frustration.	Yes	Yes
	23	Selecting text	Requires additional script development and leads to script developer frustration.	Yes	N/A
2	16	(Image recognition) Failure	Can lead to false positive test results.	N/A	N/A
	17	(Image recognition) Performance intensive	Can lead to false positive test results if image recognition affects SUT performance/timing.	N/A	N/ A
	18 (*)	(Image recognition) differentiation	Can lead to false positive test results.	N/A	N/A
	19	Animated components	Leads to false positive test results if image recognition is slower than the animation.	No	No
3	24	VNC	Leads to lowered image recognition success rate and more false positive test results.	N/A	N/A
4	2	Manual test faulty	Leads to faulty test case results if used as specification for test script or manual testing.	Yes	Yes
	3	Manual test ambiguous	Can lead to faulty test case results if used as specification for test script or manual testing.	Yes	Yes
	4	Manual test out of date	Leads to faulty test case results if used as specification for test script or manual testing.	Yes	Yes
	13	Wrong test specification	Can lead to faulty test case results if used as specification for test script or manual testing.	Yes	Yes
5	1	Missing SUT functionality	Can lead to incomplete/incorrect test case results.	Yes	Yes
	15	Simulator missing functionality	Can lead to incomplete/incorrect test case results.	Yes	Yes
	5	Missing (Missing/wrong bitmap) GUI components	Can lead to incomplete/incorrect test case results.	No	No
6	6	SUT/Tool synchronization	Can lead to faulty test case results due to incorrect timing.	Yes	N/A
	7	SUT slower after time	Can lead to faulty test case results due to incorrect timing.	Yes	N/A
	18 (*)	(Image recognition) differentiation	Can lead to false positive test results due to faulty GUI object similarity.	N/A	N/A

Table 6 Indicative analysis of how the six CPL themes from Table 5 are perceived common to other GUI-based test techniques and manual testing. For each theme, the Tier 3 CPLs from Figure 5 have been listed together with the most notesable effects of the CPLs. **(*)** - Note that CPL 18 has been listed twice because it overlaps between theme 2 and 6. The table should be interpreted as follow: **(Yes, Yes)** - The VGT CPL is also a CPL for GUI-based test techniques and manual testing. **(Yes, No)** - The VGT CPL is also a CPL for GUI-based test techniques but not for manual testing. **(No, No)** - The VGT CPL is not a CPL for GUI-based test techniques or manual testing. **(Yes, N/A)** - The VGT CPL is also a CPL for GUI-based test techniques but not applicable for manual testing. **(N/A, N/A)** - The VGT CPL is not applicable for GUI-based test techniques or manual testing.

document the test suite architecture, the functionality of help scripts and methods, it is possible to build a repository of reusable and simple to use Sikuli script code. Hence, another best practice taken from traditional programming. This practice makes it simpler for new testers and/or developers to start working with the test suite, which mitigates degradation of the test suite over time [Berner et al(2005)Berner, Weber, and Keller], improves reusability of the code and improves maintainability.

Experience reports from the study indicate that documentation and code comments are mainly required for the test suite architecture, whilst the VGT scripts are intuitive on their own. This intuitiveness comes from the combination of the GUI level operations in the scripts and the images in the scripts that define on which GUI components the operations are performed. In addition, if the scripts are implemented as 1-to-1 mappings of the manual test cases, the manual test cases also serve as documentation for the scripts.

The third general solution that was identified aims to raise script robustness and regards the removal, or non-usage, of remote computer control software, e.g. virtual network connection (VNC) applications. Removing 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. It was found in Case 2 that removing VNC can improve script success rate from 70 to 100 percent.

However, this practice has drawbacks because, even though it raises robustness, it restricts the number of test cases that can be performed on a distributed system. In addition, by running the test scripts locally, more load is put on the computer running the SUT. Hence, raising the risk of faulty SUT behavior due to a lack of performance resources, e.g. access to the central processing unit or the computers random access memory. Consequently, the use of VNC allows VGT tools, e.g. Sikuli, to test distributed systems but it lowers the success rate of the image recognition. This discussion only relates to VGT since other techniques, e.g. property-based GUI testing, require local access to the tested applications' GUI components or the layers below the GUI.

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, as stated, is not VGT specific [Horowitz and Singhera(1993)] and in addition to being a huge source of frustration during the VGT transition, it was also considered one of the most time consuming. However, the study participants reported that the CPL could be mitigated through systematic insertion of delays in the scripts. No pattern could be identified where to put the delays, instead they were placed ad hoc based on experience the study participants gained throughout the VGT transition. In Case 1, this solution practice was further supported through custom methods with an additional time delay parameter. For instance, the click(img) method from Sikuli's API, where "img" is the sought image, was expanded to click(img, delay) which delayed the script execution "delay" number of seconds before performing the click. Thus lowering the number of lines of code in each individual script and perceivably mitigated the synchronization CPL for the script developer.

However, whilst the custom methods provided additional script robustness, they also increased script execution time. The reason was because these methods required the sought image to be found twice, first by Sikuli's "wait for image method" and second by the "click method". Thus, doubling the minimum number of required image recognition sweeps. However, due to the increased robustness, reports from Case 1 still indicates that they were 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. The solution perceivably also mitigates some of the frustration related to the synchronization CPL by systematically adding dynamic delays.

In addition to the four reported solutions there were many ad hoc solutions that were too project specific to be generalized and are therefore not reported here since the manuscript is already quite substantial. Additionally there were several CPL types that could not be solved during the study. First, CPLs for which potential solutions could be identified but where the solution implementation required so much effort that it was out of scope for the project. Second, CPLs where no solution was identified at all, e.g. how to ensure image recognition robustness.

However, we believe that more general solutions can be identified and that there are several approaches to identify them. First, more case studies can be performed to empirically identify solutions. Second, since it has been established that there are many commonalities between VGT and other techniques, solutions could perceivably be migrated from said other techniques to VGT. For instance through experimentation, case studies and action research.

4.6 Defect finding ability, development cost and return on investment (ROI)

Thus far, we have focused on the CPLs related to the transition and usage of VGT, and the number of CPLs has 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 but that some operations were 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 VGT scripts. In Case 1, six defects were identified in the SUT.

1. Switching between military and civil landing lights did not work.
2. The button to switch military and civil landing lights did not disable as intended.
3. The button to switch military and civil landing lights did not disappear as intended.
4. Switching quickly between runways caused the tested system to freeze.

Nr.	Title	Description	Solves/ mitigates CPL themes
1	Failure/ exception handling	Adding failure mitigation code in the scripts, i.e. exception handling, on several levels of script abstraction can mitigate failure due to tool or image recognition volatility as well as unexpected system behavior. However, since it might be hard to predict the failures, this practice can be time consuming. For more experienced VGT script developers it is however easier to see patterns of when certain mitigation practices are required.	2 and 5
2	VGT script documentation	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/testers to start working with the scripts. In addition, this documentation should include descriptions of native Sikuli methods to complement the official, but lacking, Sikuli documentation.	1
3	Local/Remote script execution	Executing the VGT scripts without a remote system control application, e.g. virtual network control (VNC) application, raises script execution stability and image recognition robustness. However, this also limits what test cases can be executed for distributed systems. Another drawback is that the image recognition is performance intensive and can therefore modify the behavior of the tested system when run locally.	3
4	Systematic SUT synchronization	The VGT suite needs to be synchronized with the SUT using static or dynamic delays. By creating custom methods that take timing input and usage of the "wait for image method" in Sikuli script, a process can be constructed to iteratively synchronize the script with the tested system.	6
5	Other or no solution	For several of the CPLs ad hoc solutions had to be identified. These solutions included, but were not limited to, analysis and replacement of the used manual test specifications, 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. stopping Sikuli scripts from getting corrupted. Hence, there are CPLs that should be investigated in future work to be solved and/or mitigated.	1-6

Table 7 A summary of general solutions that were identified for the CPL themes presented in Table 5.

5. Tabs for switching between views would not load.
6. Logging out of Windows caused the tested system to freeze (The main thread of the tested system would not terminate).

These defects were reported 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 identified defects found that four of the defects were previously known to the company and had already been corrected in later versions of the system, but two of the defects were still unknown. Analysis of the manual test cases showed that they could identify all six defects but since two of the defects were sporadic they required several test suite executions to trigger the SUT's faulty behavior. This was also the reason why they had previously not been found since the manual tests were only executed once every test iteration. As such, the VGT scripts were able to provide the same level of confidence as the manual tests and even identify more defects.

In Case 2, the industrial practitioners reported corroborating results, i.e. the VGT scripts could identify all the defects that the manual tests could identify. Additionally, they reported that three previously unknown defects were found. In the past, due to budget constraints, not all of the system's manual test cases could be executed each iteration. However, because of the VGT transition it became possible to test more often and therefore not only provide quicker feedback to the developers but also execute more test cases for higher test coverage. In addition, the test cases could be executed with different orderings, which

Project	Dev. time (mh)	Nr. of Man. test suites	Dev. time per suite (mh)	Mainte-nance (mh)	Manual suite Exe. time (mh)	VGT suite Exe. time (h)	Pos. ROI after execu-tions
Case 1	213	1	213	-	16	15	14
Case 2	1032	3	344	266	80	2.5	13

Table 8 Summary of quantitative metrics collected during the VGT transition projects in Case 1 and Case 2. **mh** - man-hour, **h** - hour

resulted in the VGT suite finding the previously unknown defects. Thus showing the importance of what order the individual test cases are executed.

VGT provides you with the flexibility to change the execution order. However, this flexibility requires that the test cases are 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 [Cheon and Leavens(2006)]. Failure to rollback the system can result in testing of invalid system states and false negative test results. The manual test cases in Case 2 were not completely independent, which required several manual test cases to be merged. Thus, deviating from the 1-to-1 mapping between manual and automated test cases as reported in Section 3.3.

Table 8 summarizes some of the cost metrics that were acquired in Case 1 and 2. However, we stress that the contexts and conditions were significantly different between the two cases so any comparison between them need to be made with care. Whilst Case 1 was performed with only one researcher, Case 2 was performed by two industrial practitioners. In addition, the researcher in Case 1 had limited VGT and domain knowledge and knowledge of the SUT whilst the practitioners in Case 2 were domain experts for their SUT. Furthermore, only one manual test suite was transitioned into VGT in Case 1, whilst in Case 2 a total of three suites were transitioned. 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. The test suite from Case 1 was perceived to be equal in complexity to one of the more complex test suites in Case 2. Complexity was evaluated based on the qualitative and quantitative information gathered during the study about the manual test cases' number of test steps, importance of test case success, complexity of test cases GUI interactions, etc.

Furthermore, 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, as described in Section 3.4. This use case-based structure perceivably improved the manual test cases usability, maintainability and reusability since the use cases could simply be switched out to test newly added system functionality, provided that the pre- and postconditions of the new use case were aligned with the scenario. 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. Whilst the benefits such as reusability and maintainability are kept intact since new scripts can easily be created by reusing the individual use cases together with newly developed scripts. As such, this type of structure is recommended as a base for the VGT test suite architecture.

Table 8 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 that represent the actual time spent on script development in rigorous detail. In contrast, in Case 2 the development time had 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 8 comes from two different contexts and should therefore not be compared directly without taking the context, and measurement methods, into account. As such, no general conclusions can be drawn from the data. Instead, the data provides a broader view of the likely, but not yet validated, transition costs in industrial practice.

Table 8 does however show some interesting, and comparable, data regarding the improved execution time between the manual test suites and the developed VGT suites. In Case 2, the improved execution

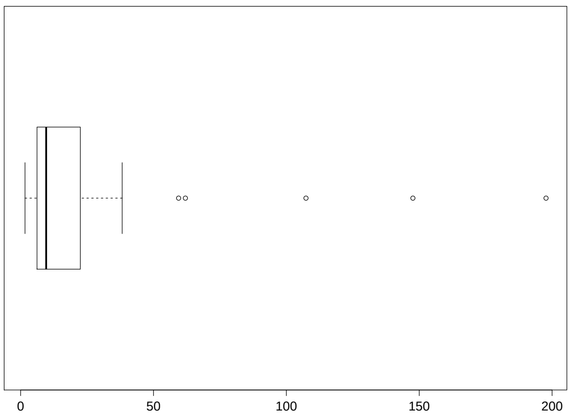

Fig. 7 Boxplot showing the execution times for individual test scripts from the VGT suite developed in Case 1. Time, shown on the x-axis, is measured in minutes.

time was quite considerable, i.e. by a factor of 16 [5], 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 but grouped together into one manual test case to save space. Performing these tests manually is very time consuming and tedious and are therefore often only performed on a few buttons 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, even these automatic tests 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 overall execution time of the test suite. Hence, some of the VGT scripts had considerably longer test execution time than others, visualized in a boxplot in Figure 7. The boxplot shows that the average execution time of the VGT scripts was 27 minutes, with a standard deviation of approximately 44. The cause of this deviation can be found in Figure 7 which shows that there were several outliers with much higher execution time, i.e. at most 198 minutes, which also affected the total execution time of the VGT suite. In addition, the execution time of the manual test suite in Case 1, 16 hours, is an ideal time, i.e. when performed by an expert tester who are capable of determining, for instance, which buttons it is necessary to repeat all the steps of the looped test cases. For a junior tester, the manual test suite can take upwards of 40 hours, as presented in our previous work [Börjesson and Feldt(2012)]. Thus, if the outliers are removed and the calculations are performed with the manual execution time of a junior tester, i.e. 40 hours, the results from Case 1 show that the automated scripts execute approximately 4.5 times faster than the manual tests.

The slow script execution time in Case 1 was also contributed by manual test cases that 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 30 seconds before triggering an alarm. Hence, for 30 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 had 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 quicker than the tested system can respond. In contrast, the test cases that were transitioned 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.

[5] Manual test execution time was 80 man hours but performed by two testers, i.e. 40 work hours in total.

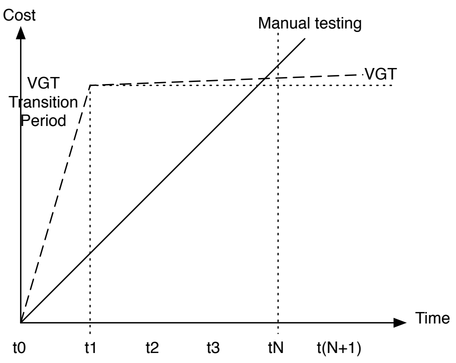

Fig. 8 Graph showing a conceptual model of when positive return on investment (ROI) would be reached after the VGT transition in a generic company. The model assumes that each manual test suite execution has a linear cost and that the VGT suite maintenance costs are low.

However, even though the tests were implemented in a 1-to-1 fashion, results from Case 1 and 2 showed that the VGT suites had improved execution time compared to the manual test suite (even though only very slightly better in Case 1). One factor that explains this speedup is that a human tester continuously has to read the manual test step descriptions 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. Furthermore, 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 these small gains add up and become significant.

Based on the collected metrics, the return on investment (ROI) for the VGT transition can be calculated by comparing the development time of the VGT suites with the manual test suites' execution time. Hence, the ROI can be calculated using the following simple equation where cost is measured in time:

$$R_{VGT} = C_{VGT}/C_{manual}$$

where R_{VGT} is the number of VGT suite executions required for the VGT suite development cost (C_{VGT}) to break even with the cost of performing the system tests manually (C_{manual}), i.e. the number of runs required to get positive ROI. This ROI calculation is however simplified, since it does not take the number of found defects into consideration, nor the costs for maintenance of the VGT suites. Maintenance costs that are currently unknown. Only initial data has been acquired [Alegroth et al(2013a)Alegroth, Feldt, and Olsson], and more studies that focus on these costs are therefore required in the future.

Once again, these result calculations are only comparable between Case 1 and 2 if the contexts of these cases are taken into account. As shown in Table 8, 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 associated with 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 [Berner et al(2005)Berner, Weber, and Keller]. Hence, both VGT suites would reach a positive ROI within less than one calendar month, given that they identify defects.

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 8, supported by the model defined by Berner et al., 2005 [Berner et al(2005)Berner, Weber, and Keller]. 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 8. It should also be noted that during the VGT transition period, t0 to t1, the VGT suite is executed continuously during development, whilst the manual test suite might not be. However, the ROI calculations assume that the VGT suite must first be completed before execution.

5 Discussion

Our findings suggests that Visual GUI Testing (VGT) is industrially applicable, can find defects as effectively as manual testing and makes economic sense even though there are many challenges and problems in transitioning to and using VGT. A majority of the identified CPLs are similar to ones that have been identified for other automated testing techniques although some are unique to VGT. Our findings imply that VGT is industrially applicable today but that there is plenty of potential to improve both the technique itself as well as its tools. Below we discuss our major findings in more detail and point to important future work before concluding our discussion with a description of the limitations of our study.

5.1 Challenges, problems and limitations

Even though a significant number of CPLs regarding the transition to or usage of VGT were identified in this study, our main conclusion is still that VGT is a valuable and cost-effective technique with equal or even better defect-finding ability as manual testing. This conclusion was corroborated in discussions with the industrial practitioners that took part in the study and contributed to the identification of the CPLs; they also consider the benefits of VGT outweighing its drawbacks. The CPLs we found originated either from the SUT, the VGT tool or the test environment and resulted in frustration and confusion during test script development, raised VGT transition costs, etc.

The CPLs and the information we have provided about them provide decision support for industrial practitioners that aim to implement VGT. Our list of CPLs can be used to anticipate problems and costs and help create plans for the transitions as well as for finding solutions in a new company and automation context. This contribution is of particular interest since it has been established that there is an industrial need and want for VGT [Alegroth et al(2013b)Alegroth, Nass, and Olsson]. In addition, the CPLs can help guide future research both in developing solutions and work-arounds but also in further development of testing techniques and tools that overcome them.

The CPLs also provide a general contribution to test automation since many of them are also likely to be encountered when transitioning to other test automation techniques. A recent systematic literature review and practitioner survey [Rafi et al(2012)Rafi, Moses, Petersen, and Mantyla] found that there is currently a lack of industrial experience reports from using test automation; our study is one step towards filling this gap. In particular, the CPLs in theme number 4 (”SUT Testability” in Table 5) are general enough to pose problems for most test automation in industrial software development, and CPLs in theme number 5 (”Test specification” in Table 5) for testing in general. Any testing activities are constrained by the correctness of the requirements and test specifications and the alignment between them and the actual and current SUT. Thus, faulty, ambiguous or incorrect test specifications lead to faulty test results regardless if they are executed manually or automatically through a script. If a SUT has not been designed or implemented to a degree where it is easily testable it is harder or sometimes impossible to adapt a testing tool to the specific SUT. Thus, resulting in incomplete test cases and additional transition costs for said technique or tool.

Another CPL that would affect most automated test techniques [Horowitz and Singhera(1993)] is ’SUT/Tool syncronization‘, e.g. how to effectively synchronize SUT and VGT script execution to avoid script failure or false positive test results. Since one of the benefits of automation is that tests can be executed faster than manual this is likely to be a problem for many automation techniques. This CPL was also reported as one of the most frustrating and costly CPLs during this study. Another ‘Script tuning’ CPL was also considered costly but is unique to VGT: how to set the similarity level of sought GUI object images in the scripts to maximize robustness. All of the CPLs unique to VGT was related to the image recognition algorithm and how to robustly ensure that the right images are matched at the right time. Future work needs to look further into the long-term effects of these VGT-unique CPLs; if the scripts need to be continuously tuned even for minor GUI changes this will affect the long-term cost effectiveness of VGT.

Several of the CPLs (theme 1 in Table 5) relates to the specific VGT tool we used, Sikuli. Sikuli is still in active development and only released as a beta version and on both of our industrial cases it was found to be immature which caused several CPLs. Since we have only used one specific tool it is also possible that we have only uncovered a subset of VGT-unique CPLs. Future work with other VGT tools could provide additional information.

5.2 Solutions

It was reported during the study that almost all of the identified CPLs could be solved using context dependent or case specific practices or technical solutions. However, only four of these solutions were found to be general beyond the studied contexts but they were also identified solve almost half of the identified CPLs whilst the retaining half required more context dependent solutions.

The observation that almost all CPLs could be solved/overcome is significant since no CPL was identified that prohibits, or renders VGT useless, in the industrial contexts we studied. However, we also found that some solutions were both time consuming and tedious. For example, much efforts had to be spent to implement a custom, yet robust, script suite solution in Sikuli (to run different selections of individual test cases in groups) and to find ad hoc solutions and work-arounds for CPLs that the practitioners had expected the tool to solve, e.g. that Sikuli cannot type Swedish letters. These two results, in particular, show the immaturity of the Sikuli tool and presents features that industrial practitioners expect from all VGT tools.

The similarities between VGT and other GUI-based test techniques makes it likely that solutions can be migrated such techniques. For instance the SUT and VGT script execution was synchronized through static and dynamic delays, as presented in [Horowitz and Singhera(1993)], in Case 1 by implementing custom methods on top of the basic Sikuli delay functionality. More research is however required to find more of these common solutions, which could be identified through literature review and verified through empirical case studies, experiments or action research.

Furthermore, even though no CPLs were identified that prohibit the use of VGT it is possible that not all CPLs can be solved in another context. One example would be how to ensure that the test specifications used for the VGT transition are unambiguous, correct and valid for the SUT. This problem is not VGT or even test related, but rather product management and development process related. Still it was perceived as a very complex problem in both our cases with complex systems with many different versions and variants. As such, it is an example of a CPL that is perceived not to be solvable in a VGT transition project but which would heavily affect the transition.

In summary, even though it was reported that most CPLs could be solved, the significant number of CPLs still post concerns for the long-term applicability of the technique and could, despite the several beneficial properties of the technique, discourage industrial practitioners from using or even testing VGT.

5.3 Defect finding ability

Results from the studied cases showed that the VGT scripts could identify all defects found by the manual test scenarios the scripts were transitioned from, but also previously unknown defects (Five in total in the studied cases). Thus, providing VGT scripts with the same level of confidence as manual test cases.

The reason why VGT can find more faults is because of the techniques low execution cost and speed. These properties allows VGT scripts to be run more often and with different orderings among the test cases. Thus stimulating more combinations of test scenarios that can occur during practical usage of the SUT and thereby uncover more defects [Rothermel et al(2001)Rothermel, Untch, Chu, and Harrold]. Additionally, some defects are intermittent and therefore require multiple test executions to be identified, which is supported by VGT. This adds to the cost-effectiveness, value and defect-finding ability of the technique and can lead to higher quality software and lower overall development costs. However, we stress that VGT, regardless of the high level of confidence, should not be used as a substitute to manual testing, rather it should be a complement. The reason is because VGT scripts, like all script-based techniques, can only find defects explicitly asserted by the test scenario [Berner et al(2005)Berner, Weber, and Keller, Itkonen and Rautiainen(2005b)]. However, it seems reasonable that VGT can be used to automate scripted test scenarios while the manual testing can be focused more on exploring the system and finding difficult usage scenarios.

An analysis performed during the study also showed that VGT can fulfill the requirements set by the RTCA DO-278 quality assurance standard, which was a requirement for the technique's use in Case 1. The main reasons is because VGT can replicate manual testing in a 1-to-1 fashion, with equivalent inputs and assertions as a human tester, and because the standard puts further emphasis on what to test, not how to test it. The results of this analysis indicate that VGT, because it provides the same level of confidence as manual testing, can fulfill other standards of similar rigor as well. This is a valuable result since it implies that VGT can be used, or at least complement other test practices, in safety- and life-critical domains.

The results regarding the defect-finding ability of VGT is also contributes to the body of knowledge about the technique and strengthens the support for the technique's industrial value and applicability. We also present six defects of varying nature that were identified using VGT that provide further decision support for industrial practitioners regarding what types of defects they can expect to identify using VGT. In addition, the defects represent a benchmark for academic researchers that aim to compare their tools and techniques against VGT and other techniques in an industrial context. However, we once again stress that VGT was only compared against manual testing in this study and that it is unclear how VGT would compare to other GUI-based or automated testing techniques. This types of comparisons is an important subject for future work.

5.4 Return on Investment

Results from the study showed that the return on investment (ROI) of transitioning the manual test suites into VGT would be positive within one month after completion, assuming that the VGT suite was executed every other night, i.e. after approximately 15 executions. We therefore conclude that VGT can be both a feasible and cost effective technique in industrial practice. However, because of the limited length of our study we have not studied the long-term cost effectiveness of the technique since we lack detailed data on VGT's maintenance costs.

Furthermore, because the studies were performed in different contexts, with different input and different number of script developers, no common cost model can be formulated from the data. The presented development costs and ROI metrics provide only indicative measures that provide further decision support for industrial practitioners. In addition, the costs are only for one test suite from Case 1 and three from Case 2 respectively, whilst a larger big-bang effort, with more test suites, would raise the up-front costs further and would require more time and more test executions for these costs to pay back. Thus, the high transition costs indicates that the choice of transition approach, iterative or big-bang, is important. As shown in related research, an iterative approach can be more beneficial [Berner et al(2005)Berner, Weber, and Keller, Williams et al(2009)Williams, Kudrjavets, and Nagappan]. However, at the same time we found that test cases that covered SUT functionality that had not been completed lead to partially implemented test cases. This shows that the SUT must have reached a certain level of maturity for VGT to be applicable since the scripts, just as system tests, aim to test SUT usage scenarios in its entirety [Beizer(2002)]. As such, in order to maximize ROI the script developer should have good knowledge about the system and perceivably start by implementing the test cases that are the most stable, test the core or most problematic functionality of the SUT. This is supported in discussions with the practitioners in Case 2 that in the future will prioritize and select test suites for transitioning based on several of these factors.

Furthermore, it should be noted that our ROI calculations do not explicitly cover maintenance costs of the VGT scripts. Maintenance was conducted in Case 2 during the study [Alegroth et al(2013a)Alegroth, Feldt, and Olsson], and has affected our ROI analysis, but in Case 1 the ROI is solely based on the development costs. As such, the calculated ROI may not be representative for a project with extensive maintenance or one with a GUI that makes image recognition harder and thus requires more script tuning.

5.5 Similarities between VGT and other GUI-based techniques

VGT has several similarities to other GUI-based test techniques such as required up front development costs, dependencies on correct SUT documentation, requirements on SUT maturity, etc. The main difference identified in this study revolve around CPLs related to the technique's use of image recognition and the currently unknown maintenance costs. These distinctions prohibit any conclusion to be drawn if VGT is better or worse than other GUI-based test techniques in industrial practice due to lacking empirical data.

The identified commonalities are however significant since it allows CPL solutions to be migrated to and from other techniques. Furthermore, this could lead to joint research opportunities and indicates that an empirical comparison between VGT and other GUI-based techniques, e.g. Record and Replay with Selenium [Holmes and Kellogg(2006)], is both feasible. However, to the authors' best knowledge, there are no studies that compare VGT with other GUI-based techniques in an industrial context; this can be an important area of future work.

As also stated, this study was only performed with one VGT tool and it is therefore possible that greater differences could be observed with other tools, e.g. JAutomate [Alegroth et al(2013b)Alegroth, Nass, and Olsson]. However, based on the commonalities between VGT tools [Alegroth et al(2013b)Alegroth, Nass, and Olsson], this is considered a minor risk.

5.6 Threats to validity

One of the limitations of this study is that both of the VGT transitions were performed on test cases for safety-critical systems with legacy code and similar characteristics. This is a threat to external validity. In particular, the results might not be applicable for smaller systems or companies that are young or immature in their development practices and processes. However, the studied 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 manual test suites at the two companies. We have mitigated this threat by analyzing and discussed 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 [Börjesson and Feldt(2012)] so we don't think this threat external validity is 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.

The study participants in Case 1 and 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 researchers 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.

We also stress that Table 6 is based on an indicative analysis based on empirical and secondary data from literature that only aims to provide an overview of the commonalities between VGT CPLs and CPLs related to other GUI-based test techniques and manual testing. As such, there may exist exceptions to the presented results. We have covered this threat by stating the tables limitations in Section 4.4.

There is additional threats to validity in Case 2 since it was driven by industrial partitioners 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.

6 Conclusions

Many software development companies rely on manual test practices to perform high-level (typically GUI-based) testing even though these practices are considered costly, tedious and error prone. Test automation, such as unit testing or Record and Replay, is generally proposed as a panacea to these types of problems. However, even though there is evidence that these techniques can be applied there is also empirical evidence to suggest that these techniques have problems that can limit their industrial applicability in certain contexts. Because of these limitations there is a need for further research into high-level test automation.

Visual GUI Testing is one such and a fairly recent technique with interest from industry. It combines image recognition with scripting to automate high-level tests. Empirical studies have shown that the technique can be applied for testing complex software systems but like any other technique, VGT has challenges, problems and limitations (CPLs) when applied on a larger scale in industrial practice. These CPLs have previously not been explored and there is a general lack of empirical studies in industry on high level and GUI level testing.

In this paper we have presented an empirical study performed in two industrial projects where researchers and industrial practitioners used VGT to automate circa 300 high-level (GUI level) system test cases in a total of four different test suites. During the study, a total of 58 CPLs were identified that were categorized into 26 unique types of CPLs that relate either to the transition to, or usage of, VGT in industrial practice. The CPLs concern different aspects: version of the tested system, the tested system in general, defects in the tested system, the company's specific hindrances, the test environment, the VGT tool's image recognition, deficiencies in the tool, or third party software.

The most prominent CPLs was classified into six main themes, three of which were related to VGT or more specifically the used VGT tool Sikuli, e.g. image recognition volatility, immaturity of the VGT tool, lack of integration with third party software and more. Two of the remaining three themes were general for any type of testing, e.g. negative effects due to ambiguous or faulty test specifications, lack of version control of the SUT, and more. The last theme was general for GUI-based testing and amounted to the need to fine-tune test scripts to ensure robust script execution. This CPLs theme was also the cause of the most frustration and cost and included how to effectively synchronize VGT script execution with the SUT's execution and how to set image similarity to make test execution robust. The identified CPLs may discourage industry practitioners from trying or using the technique, whilst also posing concerns for the long-term applicability of VGT in industrial practice, but can also be used in decision support and planning as well as in mitigating problems.

Four generic solutions that address about half of the identified CPLs, and mitigate their negative effects was identified during the study. More specific solutions were identified, but these were ad hoc and context dependent and could therefore not be generalized from the specific companies or the used tool. The latter solutions included specific development practices, additional script logic to mitigate faulty tool behavior, explicit choices of support software, etc.

Despite the identified challenges the industrial practitioners considered VGT to be both effective at finding defects and cost-effective overall. In the case project driven by industrial practitioners, the system test frequency was increased from once every six months to several times a week at a minimal cost. Three previously unknown defects were also identified during the transition project. Similar results were acquired from the case project driven by the research team where four previously known and two unknown defects were identified. These results 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 testing, whilst still providing equal or even better defect finding ability than manual testing.

In conclusion, this study has shown that VGT is a valuable and cost-effective technique for high-level (GUI level) test automation but also that it has many challenges that warrant future research. Our detailed description and analysis of the cases, the identified problems as well as solutions, costs and identified defects can help inform industrial VGT transition projects as well as help researchers prioritize their improvement efforts of VGT, GUI-based testing as well as test automation in general.

Acknowledgment

The authors of this manuscript 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. We would also like to thank the reviewers and the editor for the constructive and valuable feedback on the manuscript.

References

[Adamoli et al(2011)Adamoli, Zaparanuks, Jovic, and Hauswirth] Adamoli A, Zaparanuks D, Jovic M, Hauswirth M (2011) Automated GUI performance testing. Software Quality Journal pp 1–39

[Alegroth et al(2013a)Alegroth, Feldt, and Olsson] Alegroth E, Feldt R, Olsson H (2013a) Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, IEEE, pp 56–65

[Alegroth et al(2013b)Alegroth, Nass, and Olsson] Alegroth E, Nass M, Olsson H (2013b) JAutomate: A Tool for System- and Acceptance-test Automation. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, IEEE, pp 439–446

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

[Beizer(2002)] Beizer B (2002) Software testing techniques. Dreamtech Press

[Berner et al(2005)Berner, Weber, and Keller] Berner S, Weber R, Keller R (2005) Observations and lessons learned from automated testing. In: Proceedings of the 27th international conference on Software engineering, ACM, pp 571–579

[Börjesson(2010)] Börjesson E (2010) Multi-Perspective Analysis of Software Development: a method and an Industrial Case Study. CPL

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

[Cadar et al(2011)Cadar, Godefroid, Khurshid, Pasareanu, Sen, Tillmann, and Visser] Cadar C, Godefroid P, Khurshid S, Pasareanu CS, Sen K, Tillmann N, Visser W (2011) Symbolic execution for software testing in practice: preliminary assessment. In: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, pp 1066–1071

[Chang et al(2010)Chang, Yeh, and Miller] Chang T, Yeh T, Miller R (2010) GUI testing using computer vision. In: Proceedings of the 28th international conference on Human factors in computing systems, ACM, pp 1535–1544

[Cheon and Leavens(2006)] Cheon Y, Leavens G (2006) A simple and practical approach to unit testing: The JML and JUnit way. ECOOP 2002Object-Oriented Programming pp 1789–1901

[Dustin et al(1999)Dustin, Rashka, and Paul] Dustin E, Rashka J, Paul J (1999) Automated software testing: introduction, management, and performance. Addison-Wesley Professional

[Ebert(2007)] Ebert C (2007) The impacts of software product management. Journal of Systems and Software 80(6):850–861

[Finsterwalder(2001)] Finsterwalder M (2001) Automating acceptance tests for GUI applications in an extreme programming environment. In: Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Citeseer, pp 114–117

[froglogic(2013)] froglogic (2013) Squish. URL http://www.froglogic.com/squish/GUI-testing/

[Gamma and Beck(1999)] Gamma E, Beck K (1999) JUnit: A cook's tour. Java Report 4(5):27–38

[Grechanik et al(2009a)Grechanik, Xie, and Fu] Grechanik M, Xie Q, Fu C (2009a) Creating GUI testing tools using accessibility technologies. In: Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on, IEEE, pp 243–250

[Grechanik et al(2009b)Grechanik, Xie, and Fu] Grechanik M, Xie Q, Fu C (2009b) Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, IEEE, pp 9–18

[Grechanik et al(2009c)Grechanik, Xie, and Fu] Grechanik M, Xie Q, Fu C (2009c) Maintaining and evolving GUI-directed test scripts. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, IEEE, pp 408–418

[Gutiérrez et al(2006)Gutiérrez, Escalona, Mejías, and Torres] Gutiérrez JJ, Escalona MJ, Mejías M, Torres J (2006) Generation of test cases from functional requirements. A survey. In: 4§ Workshop on System Testing and Validation

[Hackner and Memon(2008)] Hackner DR, Memon AM (2008) Test case generator for GUITAR. In: Companion of the 30th international conference on Software engineering, ACM, pp 959–960

[Holmes and Kellogg(2006)] Holmes A, Kellogg M (2006) Automating functional tests using selenium. — pp 270–275

[Horowitz and Singhera(1993)] Horowitz E, Singhera Z (1993) Graphical user interface testing. Technical eport Us C-C S-93-5 4(8)

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

[Hsia et al(1997)Hsia, Kung, and Sell] Hsia P, Kung D, Sell C (1997) Software requirements and acceptance testing. Annals of software Engineering 3(1):291–317

[Itkonen and Rautiainen(2005a)] Itkonen J, Rautiainen K (2005a) Exploratory testing: a multiple case study. In: Empirical Software Engineering, 2005. 2005 International Symposium on, p 10 pp., DOI 10.1109/ISESE.2005.1541817

[Itkonen and Rautiainen(2005b)] Itkonen J, Rautiainen K (2005b) Exploratory testing: a multiple case study. In: 2005 International Symposium on Empirical Software Engineering, 2005., IEEE, p 10

[Leitner et al(2007)Leitner, Ciupa, Meyer, and Howard] Leitner A, Ciupa I, Meyer B, Howard M (2007) Reconciling manual and automated testing: The autotest experience. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, IEEE, pp 261a–261a

[Li and Wu(2004)] Li K, Wu M (2004) Effective GUI testing automation: Developing an automated GUI testing tool. Sybex

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

[Memon(2002)] Memon A (2002) GUI testing: Pitfalls and process. IEEE Computer 35(8):87–88

[Memon et al(2003)Memon, Banerjee, and Nagarajan] Memon A, Banerjee I, Nagarajan A (2003) GUI ripping: Reverse engineering of graphical user interfaces for testing. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), pp 260–269

[Memon and Soffa(2003)] Memon AM, Soffa ML (2003) Regression testing of GUIs. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 28, pp 118–127

[Miller and Collins(2001)] Miller R, Collins C (2001) Acceptance testing. Proc XPUniverse

[Mongrédien et al(2006)Mongrédien, Lachapelle, and Cannon] Mongrédien C, Lachapelle G, Cannon M (2006) Testing GPS L5 acquisition and tracking algorithms using a hardware simulator. In: Proceedings of ION GNSS, pp 2901–2913

[Myers et al(2011)Myers, Sandler, and Badgett] Myers G, Sandler C, Badgett T (2011) The art of software testing. Wiley

[Olan(2003)] Olan M (2003) Unit testing: test early, test often. Journal of Computing Sciences in Colleges 19(2):319–328

[Onoma et al(1998)Onoma, Tsai, Poonawala, and Suganuma] Onoma A, Tsai W, Poonawala M, Suganuma H (1998) Regression testing in an industrial environment. Communications of the ACM 41(5):81–86

[Rafi et al(2012)Rafi, Moses, Petersen, and Mantyla] Rafi D, Moses K, Petersen K, Mantyla M (2012) Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: Automation of Software Test (AST), 2012 7th International Workshop on, pp 36 –42, DOI 10.1109/IWAST.2012.6228988

[Regnell and Runeson(1998)] Regnell B, Runeson P (1998) Combining scenario-based requirements with static verification and dynamic testing. In: Proceedings of the Fourth International Workshop on Requirements Engineering-Foundations for Software Quality (REFSQ98), Pisa, Italy, Citeseer

[Rothermel et al(2001)Rothermel, Untch, Chu, and Harrold] Rothermel G, Untch R, Chu C, Harrold M (2001) Prioritizing test cases for regression testing. Software Engineering, IEEE Transactions on 27(10):929–948

[Runeson and Höst(2009)] Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2):131–164

[Sjösten-Andersson and Pareto(2006)] Sjösten-Andersson E, Pareto L (2006) Costs and Benefits of Structure-aware Capture/Replay tools. SERPS06 p 3

[smartbear(2013)] smartbear (2013) TestComplete. URL http://smartbear.com/products/qa-tools/automated-testing-tools

[TestPlant(2013)] TestPlant (2013) eggPlant. URL http://www.testplant.com/

[Vizulis and Diebelis(2012)] Vizulis V, Diebelis E (2012) Self-Testing Approach and Testing Tools. Datorzinātne un informācijas tehnoloġijas p 27

[Williams et al(2009)Williams, Kudrjavets, and Nagappan] Williams L, Kudrjavets G, Nagappan N (2009) On the effectiveness of unit test automation at Microsoft. In: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on, IEEE, pp 81–89

[Yeh et al(2009)Yeh, Chang, and Miller] Yeh T, Chang T, Miller R (2009) Sikuli: using GUI screenshots for search and automation. In: Proceedings of the 22nd annual ACM symposium on User interface software and technology, ACM, pp 183–192

[Zaraket et al(2012)Zaraket, Masri, Adam, Hammoud, Hamzeh, Farhat, Khamissi, and Noujaim] Zaraket F, Masri W, Adam M, Hammoud D, Hamzeh R, Farhat R, Khamissi E, Noujaim J (2012) GUICOP: Specification-Based GUI Testing. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE, pp 747–751