On the Industrial Applicability of Visual GUI Testing

EMIL ALGROTH
On the Industrial Applicability of Visual GUI Testing

Emil Algroth

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

Technical Report No 73L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden

This thesis has been prepared using \LaTeX.
Printed by Chalmers Reproservice,
Göteborg, Sweden 2010.
To Therese and my supporting family
Abstract

Keywords
Software Engineering, Visual GUI Testing, TBD
Acknowledgment
List of Publications

Appended papers

This thesis is based on the following papers:

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

ICST pp. 460-465.

ICST pp. 460-465.

In submission, 2012.

In submission, 2012.
Abstract

Acknowledgment

List of Publications

1 Introduction
   1.1 Introduction .................................................. 1
   1.2 Background and related work .................................. 1
   1.3 Research Focus ................................................ 1
   1.4 Methodology ................................................... 1
   1.5 Results ......................................................... 1
   1.6 Threats to Validity ............................................. 1
   1.7 Agenda for Future Research ................................... 1
   1.8 Conclusions .................................................... 1

2 Paper A
   2.1 Introduction .................................................... 4
   2.2 Research Context and the studied Company .................. 5
   2.3 The BAPO/PCF Framework ...................................... 6
   2.4 Discussion ...................................................... 11
   2.5 Conclusions ..................................................... 12

3 Paper B
   3.1 Introduction .................................................... 14
   3.2 Related Work ................................................... 15
   3.3 Case Study Description ....................................... 17
      3.3.1 Pre-study .................................................. 18
      3.3.2 Industrial Study ......................................... 19
   3.4 Results ......................................................... 21
      3.4.1 Results of the Pre-study ................................. 22
      3.4.2 Results of the industrial study ......................... 26
   3.5 Discussion ..................................................... 28
   3.6 Conclusion ..................................................... 30
4 Paper C
4.1 Introduction ........................................ 34
4.2 Related Work ........................................ 35
4.3 Research methodology ............................... 36
  4.3.1 Research site .................................. 37
  4.3.2 Research process ............................... 37
4.4 Results and Analysis ............................... 39
  4.4.1 Pre-transition .................................. 39
  4.4.2 During transition ............................... 41
    4.4.2.1 VGT test suite maintenance for improvement . 42
    4.4.2.2 VGT test suite maintenance required due to SUT change . 43
  4.4.3 Post-transition ............................... 45
4.5 Discussion ........................................ 48
  4.5.1 Threats to validity ............................ 51
4.6 Conclusion ........................................ 51

5 Paper D
5.1 Introduction ........................................ 54
5.2 Background and Related work ...................... 55
5.3 Industrial case study ............................... 57
  5.3.1 The industrial projects ....................... 58
  5.3.2 Detailed data collection in Case 1 ............ 60
  5.3.3 Detailed data collection in Case 2 ............ 62
  5.3.4 The VGT suite ................................ 63
5.4 Results and Analysis ............................... 64
  5.4.1 Test system related CPLs ....................... 65
    5.4.1.1 Test system version ....................... 65
    5.4.1.2 Test system (General) .................... 68
    5.4.1.3 Test system (Defects) .................... 71
    5.4.1.4 Test company specific CPLs ............... 73
    5.4.1.5 Test system (Environment) ............... 73
  5.4.2 Test tool related CPLs ....................... 74
    5.4.2.1 Test tool (Sikuli) related CPLs .......... 74
    5.4.2.2 Test application ........................ 78
  5.4.3 Support software related CPLs ............... 80
  5.4.4 CPL Summary ................................ 81
  5.4.5 Potential CPL solutions ....................... 83
  5.4.6 Defect finding ability, development cost and return on investment (ROI) .................. 87
5.5 Discussion ........................................ 92
  5.5.1 Challenges, Problems, Limitations and Solutions .... 92
  5.5.2 Defects and performance ...................... 93
  5.5.3 Visual GUI Testing with Sikuli ............... 94
  5.5.4 Threats to validity .......................... 95
5.6 Conclusions ...................................... 96
6 Paper E

6.1 Introduction .................................................. 100
6.2 Related Work .................................................. 101
6.3 JAutomate ....................................................... 102
  6.3.1 Tool comparison ........................................... 103
  6.3.2 Business ................................................... 106
  6.3.3 Architecture ............................................... 106
    6.3.3.1 The tool ............................................. 106
    6.3.3.2 The system under test (SUT) ....................... 108
  6.3.4 Process ................................................... 109
  6.3.5 Organisation .............................................. 110
6.4 The industrial need .......................................... 110
6.5 Discussion .................................................... 112
6.6 Conclusion .................................................... 113

Bibliography ..................................................... 115
Chapter 1

Introduction

1.1 Introduction
1.2 Background and related work
1.3 Research Focus
1.4 Methodology
1.5 Results
1.6 Threats to Validity
1.7 Agenda for Future Research
1.8 Conclusions
Chapter 2

Paper A

Structuring Software Engineering Case Studies to Cover Multiple Perspectives
E. Börjesson, R. Feldt

Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE’2011), Boston, Massachusetts, USA, July 1-3, 2009 pp. 460-465
Abstract

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

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

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

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

The case study is an observational research method used in many different fields of research due to its flexibility and its ability to investigate a phenomenon in its context [1]. Case studies are also applicable when there is no clear distinction between the phenomena and its context. This is particularly true in empirical software engineering research where there are many factors that impact the phenomenon, such as the type and organization of the company, the development processes used etc. To understand a contemporary phenomenon we also need to understand its history and how the different factors have evolved over time.

Existing advice for empirical research in software engineering focus on experiments and systematic reviews while guidelines for case studies was only recently published by Runeson and Höst [1]. There, a high quality case study is defined as a study that produces valid information of academic or industrial significance, either generic or practitioner oriented [1]. A key criteria for achieving this is that the data is collected in a planned and consistent manner and that conclusions are based on a clear chain of evidence. This can be a challenge in practice since case studies are typically flexible research designs with multiple sources of evidence; it is not clear-cut how to find the right balance between a flexible research design that allows multiple factors and causes to be taken into account while providing enough structure and support for planning and analysis.

Software engineering is different from computer science in that it takes more perspectives than only the technical into account. For example the personality or motivation of the engineers [2, 3] can affect the quality of their work and organizational and career considerations can affect important activities such as effort estimation [4,5]. In general, the characteristics not only of the people but of the organization, business and processes used in software development are all important.

In recent empirical research in industry we were faced with designing a case study to describe and understand the verification and validation practices at a Swedish company developing safety-critical software systems. From initial talks at the company we understood that the practices could not be studied in isolation; they were heavily tied to the whole context of the company as well as how they had evolved over time. To structure our data collection and understanding we based our case study on an analytical matrix combining these two main aspects. On one dimension we wanted to cover at least the main perspectives of the BAPO framework with its four aspects [6]: Business, Architecture (technical aspects), Processes and Organization. The other dimension was time detailed in three steps as Past, Current and Future. Together this created a matrix of 12 different sub areas to be considered during the case study. We also defined a general six-step process to design, collect and analyze the findings of the study and adapted it to the company. This paper describes this analytical framework, called the BAPO/PCF framework for software engineering case studies, covering both the matrix and the process to adapt and use it. We exemplify and evaluate the framework based on our application of it at the studied company. In particular this paper addresses the following research questions:
[A] What are the perceived advantages of a multi-perspective approach to conducting a case study?

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

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

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

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

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

2.2 Research Context and the studied Company

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

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

2.3 The BAPO/PCF Framework

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

Get Knowledge about the Domain. Very little was known about the company when the project started and therefore the first step of the process was to gain a deeper understanding about the company and the company context. This information was acquired through a combination of interviews with people at the company and through literature review of internal documentation as well as documentation about the company’s domain. The context in which the company operates was important to understand the goals and needs of the company, in order to align the improvement effort with these goals, such as process changes or introduction of new tools, etc. The initial interviews also served as a way of finding new sources by using snowballing, sources that could be used later during the company study. Snowballing is conducted by asking questions at the end of interviews and looking at document references to locate further artefacts to study [8]. The majority of the domain documentation that was studied was market quality standards and frameworks. Development and quality standards were important to study because the results of the company study would be used as a base for a larger project that focused on the company’s verification and validation (V&V) practices, hence changes done to the V&V practices would, other than to comply with company needs, also have to comply with such standards. Domain specific documentation is often hard for someone from outside the domain to understand, so therefore all interpretations and question marks related to the analyzed domain documentation had to be verified with people at the company.

The collected data from step 1 showed that a broader view would be required to capture the state-of-practice at the company in its context. To meet this requirement and structure the research a multi-perspective framework was developed based around an analytical matrix with two dimensions. The first dimension was chosen to represent the perspectives of the company that would be analyzed, defined as company Business, Software Architectures, Processes and Organization (BAPO). These perspectives were chosen because they are used within academic and industrial frameworks such as the Family evaluation framework, which is used to evaluate Software Product Lines [6]. The second
dimension of the matrix was chosen to represent time, defined as the Past, Current and Future (PCF), since a chronological dimension of the elicited information was perceived to enable deeper and more precise conclusions to be drawn from the research findings. Hence the matrix starts with past Business, moving on to current business and so on, until finally reaching future organization. The BAPO/PCF framework was also chosen because it was perceived to be a good fit in this particular study, which might not be the case in the context of another company.

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

*Choice of Detailed Research Methods.* The third step of the process was to choose what research methods that would be used in the company study. Primarily qualitative methods were chosen, such as interviews, visual inspection, structured observation and water-cooler discussions. Water-cooler discussions are conducted during coffee and lunch-breaks and make use of the fact that most information sharing within development companies are conducted during breaks [9]. This allows the researcher to get qualitative data that was current and truthful, but since the information often included the persons own opinions the information had to be verified to remove bias. Surveys were also used to provide quantitative results that would give depth to the research results.
The surveys were deductive, hence based on previously elicited information, to verify the collected results and therefore conducted later in the company study. Different research methods are more appropriate to use depending on what sources are available, and therefore it is good to have a set of methods to choose from [1]. Creating a plan of what methods to use in what cells is suitable, but it should be considered a guideline that can be changed depending on the conditions of the study rather than a strict plan.

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

Case studies require flexibility to be able to respond to different events that may occur during the course of the study, such as for instance sudden research opportunities. For instance during the company study an opportunity arose to do observation of the company’s system testing practices that would not arise again during the course of the planned study period. Because events, such as opportunities, have to be considered the research has to be flexible and can not follow a strict research plan. The BAPO/PCF framework does however support such flexibility, since the research questions connected to the BAPO/PCF matrix cells can be answered out of order, which allows the researcher to jump between cells and work with the question that is the most urgent at the moment. The progress of the research can also be measured through what research questions within the matrix have been answered. Each cell that gets filled with information, and its research question answered, constitutes a measurable part of the study progress, and once all questions in all cells have been answered the study is complete. The time spent with a certain matrix cell can either be planned according to a fixed time budget, or in a more ad hoc manner where the collected results drive the time spent working with a cell as well as the order which the matrix cells are traversed. There are academic papers that state that research using the BAPO perspectives should start with the business aspect because this has the largest impact on the company, followed by software architectures that has the second largest impact and so on until the organizational aspect has finally been analyzed [6]. No evidence could be found to support this claim during the company study,
which used an approach where intermediate results decided what cell to jump to next in the study. The study at the company did not follow the matrix row by row, but instead company processes were first investigated, followed by software architectures, followed by company business and finally company organization. The BAPO/PCF matrix traversal was not chosen at random, it was chosen based on information that became available in the initial part of the forth process step and proved throughout the study to be valuable asset to keep the research focused. Even though several deviations were made from the research plan, because opportunities arose, it proved to be valuable to have a plan to fall back on, which is also supported by Runeson and Höst that state that a plan is crucial for a case study to succeed [1].

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

The BAPO/PCF matrix can also be used as a tool to visualize the research result quality. Several techniques were used to ensure data validity during the company study, but the primary technique was triangulation [1, 10], which
states that in order for information to be valid it must be verified by at least three sources. These sources can be either artefacts or roles, for instance documentation, different roles at the company, etc, or results gathered by using several research methods, such as interviews, literature review, etc. The matrix can be used to show how triangulation was performed during a study by visualizing which sources were used to answer what research question within the BAPO/PCF matrix. This provides a graphical overview of the data validity that is easy to understand, also by non-researchers like for instance the managers of the company under study. The general triangulation technique is presented in Figure 2.2, and the visualization of what sources that were used for triangulation, in the company study, is presented in Figure 3.4. Another technique that was used to ensure data validity during this step of the process, in the company study, was to send transcripts of interviews, preliminary documentation analysis reports, etc, to different roles within the studied company for validation. This technique provides the researcher with fast feedback if the elicited information has been correctly interpreted or not, which is valuable in cases where the information includes contextual jargon unknown to the researcher.

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

The perceived advantages of applying the BAPO/PCF framework for case studies are related to the structure and the broadness of the multi-perspective view of the framework. By applying the multi-perspective approach to a case study it becomes possible to collect results specific to the different perspectives of the company, as well as in time. This provides a contextual structure that makes it clear where and how to acquire information to answer the research questions, but also how to find information that can link the results together. For instance by pointing out a role within the studied company that has knowledge about several perspectives, the information gained from this knowledge can be used to draw cross-perspective conclusions about the cause of a particular result. By looking at the entire matrix, including these cross-perspective results, longer chains of cause and effect can also be drawn that provides a deeper as well as broader understanding of results and of the company. Internal validity is also improved with this approach since as described by Wohlin et al. 2000 [11], a factor that is investigated because of its effects on another factor may itself be affected by a third factor, and if the third factor is overlooked there may be a threat to internal validity. Hence the BAPO/PCF framework improves the internal validity by giving the researcher the means of finding these connections between different factors that could have otherwise been overlooked with a narrower scope. It is however important to recognize that this framework, like other case studies, does not solve the issue regarding to what extent a factor within a given perspective affects another factor in another given perspective. A change within a company is seldom localized to a certain perspective of the company but rather has ripple effects to several different perspectives. An example from the company study is how changes to a company’s business goals resulted in the introduction of new processes, new architectural development methods as well as organizational change.

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

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

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

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

2.5 Conclusions

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

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

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

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

Paper B


E. Börjesson, R. Feldt

ICST pp. 460-465
Abstract

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

Market trends with demands for faster time-to-market and higher quality software continue to pose challenges for software companies that often work with manual test practices that can not keep up with increasing market demands. Companies are also challenged by their own systems that are often Graphical User Interface (GUI) intensive and therefore complex and expensive to test [12], especially since software is prone to changing requirements, maintenance, refactoring, etc., which requires extensive regression testing. Regression testing should be conducted with configurable frequency [13], e.g. after system modification or before software release, on all levels of a system, from unit tests, on small components, to system and acceptance tests, with complex end user scenario input data [14, 15]. However, due to the market imposed time constraints many companies are compelled to focus or limit their manual regression testing with ad hoc test case selection techniques [16] that do not guarantee testing of all modified parts of a system and cause faults to slip through.

Automated testing has been proposed as one solution to the problems with manual regression testing since automated tests can run faster and more often, decreasing the need for test case selection and thereby raising quality, while reducing manual effort. However, most automated test techniques, e.g. unit testing [17, 18], Behavioral Driven Development [19], etc., approach testing on a lower system level that has spurred an ongoing discussion regarding if these techniques, with certainty, can be applied on high-level system tests, e.g. system tests [20, 21]. This uncertainty has resulted in the development of automated test techniques explicit for system and acceptance tests, e.g. Record and Replay (R&R) [22–24]. R&R is a tool-supported technique where user interaction with a System Under Test’s (SUT) GUI components are captured in a script that can later be replayed automatically. User interaction is captured either on a GUI component level, e.g. via direct references to the GUI components, or on a GUI bitmap level, with coordinates to the location of the component on the SUT’s GUI. The limitation with this technique is that the scripts are fragile to GUI component change [25], e.g. API, code, or GUI layout change, which in the worst case can render entire automated test suites inept [26]. Hence, the state-of-practice automated test techniques suffer from limitations and there is a need for a more robust technique for automation of system and acceptance tests.

In this paper, we investigate a novel automated testing technique, which we in the following call visual GUI testing, with characteristics that could lead to more robust system test automation [27]. Visual GUI testing is a script based testing technique that is similar to R&R but uses image recognition, instead of GUI component code or coordinates, to find and interact with GUI bitmap components, e.g. images and buttons, in the SUT’s GUI. GUI bitmap interaction based on image recognition allows visual GUI testing to mimic user behavior, treat the SUT as a black box, whilst being more robust to GUI layout change. It is therefore a prime candidate for better system and acceptance test automation. However, the body of knowledge regarding visual GUI testing is small and contain no industrial experience reports or other studies to support the techniques industrial applicability. Realistic evaluation on industrial scale
testing problems are key in understanding and refining this technique. The body of knowledge neither contains studies that compare different visual GUI testing tools or the strengths and weaknesses of the technique in the industrial context.

This paper aims to fill these gaps of knowledge by presenting a comparison of two visual GUI testing tools, one commercial referred to as Commercial-Tool\(^1\), and one open source, called Sikuli \([27]\), in an industrial context to answer the following research questions:

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

[B] What are the advantages and disadvantages of visual GUI testing for system regression testing?

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

The paper is structured as follows; section 3.2 presents related work followed by section 3.3 that describes the case study design. Section 4.4 presents results which are then discussed in section 6.5. Section 6.6 concludes the paper.

### 3.2 Related Work

The body of knowledge on using GUI interaction and image recognition for automation is quite large and has existed since the early 90s, e.g. Potter \([28]\) and his tool Triggers used for GUI interactive computer macro development. Other early works includes Zettlemoyer and Amant who explored GUI automation with image recognition in their tool, VisMap. VisMap’s capabilities were demonstrated through automation of a visual scripting program and the game Solitaire \([29]\). These early works did however not focus on automated testing but rather automation in general with the help of image recognition algorithms.

There is also a large body of knowledge on using GUI interaction for software testing, as shown by Adamoli et al. \([22]\) who have surveyed 50 papers related to automated GUI testing for their work on GUI performance testing. Note that we differentiate between GUI interaction for automation and GUI interaction for testing since all techniques for GUI automation are not intended for testing and vice versa.

One of the most common GUI testing approaches is Record and Replay (R&R) \([22-24]\). R&R is based on a two step process where user mouse and

\(^1\)For reasons of confidentiality we cannot disclose the name of the tool.
keyboard inputs are first recorded and automatically stored in a script that
the tool can then replay in the second step. Different R&R tools record user
input on different GUI abstraction levels, e.g. the GUI object level or the GUI
bitmap level, with different advantages and disadvantages for each level. On
the top GUI bitmap level a common approach is to save the coordinates of
the GUI interaction in a script, with the drawback that the script becomes
sensitive to reconfiguration of GUI layout but with the advantage of making
the scripts robust to API and code changes. The other R&R approach is to
record SUT interaction on a lower GUI object level by saving references to
the GUI code components, e.g. Java Swing components, which instead make
the scripts sensitive to API and code structure change [26] but more robust to
GUI layout reconfiguration.

GUI testing can also be conducted on the top GUI bitmap level with tech-
niques that use image recognition to execute test scenarios [27], in this paper
referred to as visual GUI testing. Visual GUI testing is very similar to the
R&R approach but with the important distinction that R&R tools do not use
image recognition and are thus more hardcoded to the exact positioning of
GUI elements. In current visual GUI testing tools, the common approach is
that scenarios are written manually in scripts that include images for SUT
interaction in contrast to the R&R approach where test scripts are commonly
generated automatically with coordinates or GUI component references. In a
typical visual GUI testing script input is given to the SUT through automated
mouse and keyboard commands to GUI bitmap components identified through
image recognition, output is then observed, once again with image recognition,
and compared to expected results after which the next sequence of input is
given to the SUT, etc. The advantages of visual GUI testing is that it is im-
pervious to GUI layout reconfiguration, API and code changes, etc., but with
the disadvantage that it is instead sensitive to changes to GUI bitmap objects,
e.g. change of image size, shape or color.

A different approach to GUI testing is to base it on models, e.g. generate
test cases from finite state machines (FSM) [30, 31]. However, the models
often need to be created manually at considerable cost and the approach often
face scalability problems. Automated model creation approaches have been
proposed, such as GUI ripping proposed by Memon [32].

Hence, the area of GUI interaction, automation and testing, is quite broad
but limited regarding empirical studies evaluating the techniques on real-world,
industrial-scale software systems. Comparative research has been done on tools
that use the R&R technique [22], but, to the authors’ knowledge, there are no
studies that compare visual GUI testing tools or evaluate if they can substitute
manual regression testing in the industrial context.

Another important test aspect is acceptance testing where user and cus-
tomer requirement conformity is verified with test scenarios that emulate end
user interaction with the SUT. The tests are similar to system test cases, but
contain more end user specific interaction information, i.e. how the system will
be used in its intended domain. Acceptance test scenarios should preferably
be automated and run regularly to verify system conformity to the system
requirements [13] and has therefore been subject to academic research. The
academic research has resulted in both tools and frameworks for acceptance
test automation, including tools for GUI-interaction [33], but to the authors’
knowledge there is no research using visual GUI testing for acceptance testing.

3.3 Case Study Description

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

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

The methodology used in the study was divided into two main phases, shown in Figure 3.1, with three steps in each phase. Phase one of the study was a pre-study with three different steps. An initial tool analysis compared the tools based on their static properties as evaluated through ad hoc script development and review of the tools’ documentation. This was followed by a series of experiments with the goal of collecting quantitative metrics on the strengths and weaknesses of the tools. The experiments also served to provide information about visual GUI testing’s applicability for different types of GUIs, e.g. animated with moving objects and non-animated with static buttons and images, which would provide decision support for, and possibly rule out, what type of system to study at Saab in the second phase of the study. In parallel with these experiments an analysis of the industrial context at Saab was also conducted. Phase two of the study was conducted at Saab and started with a complete manual system test of all the 50 test cases of the studied subsystem. This took 40 hours, spread over five days, during which the manual test cases were categorized based on their level of possible automation with the visual GUI testing tools. Both of the visual GUI testing tools were then used to automate five, carefully selected, representative, test case scenarios (ten percent) of the manual test suite during which metrics on
script development time, script LOC and script execution time were collected.

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

### 3.3.1 Pre-study

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

In parallel with the industrial context analysis, static properties of the studied tools were collected, through explorative literature review of the tools’ documentation and ad hoc script development. The collected properties were then analyzed according to the quality criteria proposed by Illes et al. [34], derived from the ISO/IEC 9126 standard supplemented with criteria to define tool vendor qualifications. The criteria refer to tool quality and are defined as **Functionality, Reliability, Usability, Efficiency, Maintainability, Portability, General vendor qualifications, Vendor support, and Licensing and pricing.**

The tools were also analyzed in four structured experiments where scripts were written in both tools, with equivalent instructions to make the scripts comparable, and then executed against controlled GUI input. The GUI input was classified into two groups, animated GUIs and non-animated GUIs, chosen to cover and evaluate how the tools perceivably performed for different types of industrial systems. The ability to handle animated GUIs is critical for visual GUI testing tools since they apply compute-intensive image recognition algorithms that might not be able to cope with highly dynamic GUIs. Eight scripts were written in total, four in each tool, and each one was executed in 30 runs for each experiment. The experiments have been summarized in the following list:

- **Experiment 1**: Aimed to determine how well the tools could differentiate between alpha-numerical symbols by adding the numbers six and nine in a non-animated desktop calculator by locating and clicking on the calculator’s buttons.

- **Experiment 2**: Aimed to determine how the tools could handle small graphical changes on a large surface, tested by repeated search of the computer desktop for a specific icon to appear that was controlled by the researcher.
3.3. CASE STUDY DESCRIPTION

- Experiment 3: Aimed to test the tools image recognition algorithms in an animated context by locating the back fender of a car driving down a street in a video clip in which the sought target image was only visible for a few video frames.

- Experiment 4: Also in an animated context, aimed to identify how well the tools could track a moving object over a multi-colored surface in a video clip of an aircraft, represented by its textual call-sign, moving across a radar screen.

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

The experiments were run on a MacBook Pro computer, with a 2.8GHz Intel Core 2 Duo processor, using virtual network computing (VNC) [35], which was a requirement for CommercialTool. CommercialTool is designed to be non-intrusive, meaning that it should not affect the performance of the SUT, and to support testing of distributed software systems. This is achieved by performing all testing over VNC and support for it is built into the tool. Sikuli does not have VNC support so to equalize the experiment conditions Sikuli was paired with a third party VNC viewer application. The VNC viewer application was run on one user account connected to a VNC server on a second user account on the experiment computer, visualized in Figure 3.2.

Finally the visual GUI testing tools were also analyzed in terms of learnability since this aspect affects the technique’s acceptance, e.g. if the tool has a steep learning curve it is less likely to be accepted by users [36]. The learnability was evaluated in two ad hoc experiments using Sikuli, where two individuals with novice programming knowledge, at two different occasions, had to automate a simple computer desktop task with the tool.

3.3.2 Industrial Study

The studied subsystem at Saab consisted of two computers with the Windows XP operating system, connected through a local area network (LAN). The LAN also included a third computer running simulators, used during manual testing to emulate domain hardware controlled by the subsystem’s GUI. The GUI consisted primarily of custom-developed GUI components, such as buttons and other bitmap graphics, and was non-animated. During the study a fourth computer was also added to the LAN to run the visual GUI testing tools and VNC, visualized in Figure 3.3. VNC is scalable for distributed systems so the level of complexity of the industrial test system setup, Figure 3.3, was
directly comparable to the complexity of the experimental setup used during the pre-study, Figure 3.2.

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

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

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

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

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

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

Below the results gathered during the study are presented divided into the results gathered during the pre-study and the results gathered during the in-

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

Table 3.1: Properties of the manual test cases selected for automation. The number of physical computers does not include the computer used to run the visual GUI testing tools.

[C] The number of run-time reconfigurations of the subsystem the test case included.

The number of physical computers would impose complexity by requiring additional VNC control code and interaction with a broader variety of GUI components, e.g. interaction with custom GUI components in subsystem part A and B and the simulators. Simulator interaction was also important to cover in the automated test cases since if some simulator interaction could not be automated neither could the manual test cases using that simulator. Run-time reconfiguration in turn added complexity by requiring the scripts to read and write to XML files. In Table 3.1 the five chosen test cases have been summarized together with which of the three properties they automate. The minimum number of physical computers required in any test case were two and maximum three whilst the maximum number of run-time configurations in any test case were also three. There were four simulators, referred to as A, B, C and D, but only simulators A and B were automated in any script because they were the most commonly used in the manual test cases and also had the most complex GUIs. In addition, simulators C and D had very similar functionality to A and B and had no unique GUI components not present in A or B and were therefore identified as less important and possible to automate.

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

3.4.1 Results of the Pre-study

The pre-study started with a review of the studied visual GUI testing tools’ documentation from which 12 comparable static tool properties relevant for Saab were collected. The 12 properties are summarized in Table 6.1 that shows which property had impact on what tool quality criteria defined by Illes et al. [34], described in section 3.3. The table also shows what tool was the most favorable to Saab in terms of a given property, e.g. CommercialTool was more favorable in terms of real-time feedback than Sikuli. The favored tool is represented in the table with an S for Sikuli, CT for CommercialTool and (-) if the tools were equally favorable.

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

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

*Script Language syntax.* The script language in Sikuli is based on Python, extended with functions specific for GUI interaction, e.g. clicking on GUI objects, writing text in a GUI, waiting for GUI objects, etc. Sikuli scripts are written in the tool’s Integrated Development Environment (IDE) and because of the commonality between Python and other imperative/Object-Oriented languages the tool has both high usability and learnability with perceived positive impact on script maintainability. The learnability of Sikuli is also supported by the learnability experiments conducted during the pre-study, described in Section 3.3, where novice programmers were able to develop simple Sikuli scripts after only 10 minutes of Sikuli experience and advanced scripts after an hour.

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

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

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

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

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

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

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

Remote SUT connection support / requirement. Sikuli does not have built in VNC support, a property that is not only supported by CommercialTool but also required by the tool to operate. Sikuli was therefore paired with a third party VNC application as described in Section 3.3, to provide Sikuli with the same functionality, usability and portability as CommercialTool.

Cost. The studied tools differ in terms of cost since Sikuli is open source with no up-front cost whilst CommercialTool costs around 10,000 Euros per ‘floating license’ per year. A floating license means that it is not connected to any one user or computer but only one user can use the tool at a time, hence the Licensing and pricing quality criterion in this case affects the usability of CommercialTool since some companies may not afford multiple licenses while still wanting to run multiple scripts at the same time.

Backwards compatibility and support. The last property concerns the backwards compatibility of the tools, and whilst CommercialTool’s vendor guarantees that the tool, which has been available in market for several years, will always be backwards compatible, Sikuli is still in beta testing and therefore subject to change. Changes to Sikuli’s instruction set could affect the functionality and maintainability of the tool and scripts. This property also provides general vendor qualification information, e.g. the maturity of the vendor and
<table>
<thead>
<tr>
<th>Property</th>
<th>CommercialTool</th>
<th>Sikuli</th>
<th>Impacts</th>
<th>Favored tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed in</td>
<td>C#</td>
<td>Jython</td>
<td>F/P/VSS</td>
<td></td>
</tr>
<tr>
<td>Script language syntax</td>
<td>Custom</td>
<td>Python</td>
<td>F/U/M</td>
<td>S</td>
</tr>
<tr>
<td>Supports imports</td>
<td>No</td>
<td>Java and Python</td>
<td>F/U/E/VS</td>
<td>S</td>
</tr>
<tr>
<td>Image representation in tool IDE</td>
<td>Text-Strings</td>
<td>Images</td>
<td>F/U/M</td>
<td>$F$</td>
</tr>
<tr>
<td>Real-time script execution feedback</td>
<td>Yes</td>
<td>No</td>
<td>U/M</td>
<td>CT</td>
</tr>
<tr>
<td>Image recognition sweeps per second</td>
<td>7</td>
<td>5</td>
<td>F/R/U</td>
<td>CT</td>
</tr>
<tr>
<td>Image recognition failure mitigation</td>
<td>Multiple algorithms to choose from</td>
<td>Image similarity configuration</td>
<td>F/R/U/E/M/P</td>
<td>CT</td>
</tr>
<tr>
<td>Test suite support</td>
<td>Yes</td>
<td>Unit tests only</td>
<td>F/U/M/P</td>
<td>-</td>
</tr>
<tr>
<td>Remote SUT connection support</td>
<td>Yes</td>
<td>No</td>
<td>F/U/P</td>
<td>-</td>
</tr>
<tr>
<td>Remote SUT connection requirement</td>
<td>Yes</td>
<td>No</td>
<td>F/U/P</td>
<td>S</td>
</tr>
<tr>
<td>Cost</td>
<td>10.000 Euros per license per computer</td>
<td>Free</td>
<td>U/LP</td>
<td>S</td>
</tr>
<tr>
<td>Backwards compatibility</td>
<td>Guaranteed</td>
<td>Uncertain</td>
<td>F/M/GVQ</td>
<td>CT</td>
</tr>
</tbody>
</table>

Table 3.2: Results of the property comparison between CommercialTool and Sikuli. Column **Impacts**: F - Functionality, R - Reliability, U - Usability, E - Efficiency, M - Maintainability, P - Portability, GVQ - General Vendor qualifications, VS - Vendor Support, LP - Licensing and pricing. Column **Favored tool**: S - Sikuli, CT - CommercialTool, (−) - Equal between the tools
3.4. RESULTS

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

Table 3.3: Academic experiment results. CT stands for CommercialTool. Type indicates if the experiment was non-animated or not and Desc. describes the experiment.

The second part of the pre-study consisted of four structured experiments, described in Section 3.3 and their results are summarized in Table 3.3. In the first experiment a script was developed in each tool for a non-animated desktop calculator application to evaluate CommercialTool’s and Sikuli’s image recognition algorithms’ ability to identify alpha-numeric symbols. Sikuli only had a success rate of 50 percent in this experiment, over 30 runs, because the tool was not always able to distinguish between the number 6 and the letter C, used to clear the calculator, whilst CommercialTool had a success rate of 100 percent. In the second experiment the goal was to find a specific icon as it appeared on the desktop, hence identify a small bitmap change on a large surface, for which both tools had a 100 percent success rate. In the third experiment the goal was to identify the back fender of a car driving down a road in a video clip where the sought fender image was only visible for a few video frames, imposing a time constraint to the image recognition algorithms. The car experiment resulted in Sikuli having a success rate of 25 percent and CommercialTool 3 percent. The final experiment required the tools to trace the call sign, a text string, of an aircraft moving over a multi-colored radar screen in a video-clip, where Sikuli had a 100 percent success rate whilst CommercialTool’s success rate was 0 percent.

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

The industrial part of the study started with the researchers conducting a complete manual system test of the studied subsystem. During the manual system test all the test cases were analyzed, as described in Section 3.3, and classified into categories. The category analysis showed that Sikuli could fully script 81%, partially script 17% and not script 2% of the manual test cases. CommercialTool on the other hand could fully script 95%, partially script 3% and not script 2% of the manual test cases. The higher percentage of scripts that could be fully automated in CommercialTool was given by the tool’s ability to analyze audio output, required in seven of the manual test cases. The 2% of the manual test cases that could not be scripted, in either tool, were hardware related and required physical interaction with the SUT.

Based on the categorization and the selection criteria, discussed in Section 3.3, five manual test cases were chosen for automation. The automation was done pair-wise in each tool, e.g. test case x was automated in one tool and then in the other tool, with the order of the first tool chosen at random for each test case. Random tool selection was used to ensure that the script development time for the script developed in the secondly used tool would not continuously be skewed, lowered, because challenges with the script, e.g. required failure mitigation, etc., had already been resolved when the script was developed in the first tool.

The main contributor to script development time was in the study observed to be the amount of code required to mitigate failure due to unexpected system behavior, e.g. GUI components not rendering properly, GUI components appearing on top of each other, etc. Failure mitigation was achieved through ad hoc addition of wait functions, conditional branches and other exception handling, e.g. try-catch blocks, which for each added function required extra image recognition sweeps of the GUI that also increased the script execution time. Scripts that required failure mitigation also took longer to develop since they had to be rerun more times during development to ensure script robustness. The development time required to make a script robust also proved to be very difficult to estimate because unexpected system behavior was almost never related to the test scenarios but rather a product of the subsystem’s implementation. Each script was developed individually and consisted of three parts. First a setup part to cover the preconditions of the test case. The second part was the test scenario and the third part was a test teardown to put the subsystem back in a known state to prepare it for the following test case. After the five test scripts had been developed in each tool the LOC and execution time for each script was recorded, shown in Table 3.4 together with the script development time and number of steps in the corresponding manual test case scenario.

Table 3.4 shows that the total development time, LOC and execution time were similar for the scripts in both tools.

The five chosen test cases were carefully selected to be representative for the entire manual test suite for the subsystem, as described in section 3.3, to allow the collected data to be used for estimation. Estimation based on the average execution times, from Table 3.4, shows that the fully automated test suite for the subsystem, all 50 test cases, would run in approximately three and
3.4. RESULTS

Table 3.4: Metrics collected during test case automation. CT stands for CommercialTool, ATC for automated test case and TC steps for the number of test steps in the scenario of the manual test case.

<table>
<thead>
<tr>
<th>Test case</th>
<th>CT Dev-time (min)</th>
<th>Exe-time (sec)</th>
<th>Sikuli Dev-time (min)</th>
<th>Exe-time (sec)</th>
<th>LOC</th>
<th>TC Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATC-1</td>
<td>255</td>
<td>111</td>
<td>105</td>
<td>90</td>
<td>212</td>
<td>5</td>
</tr>
<tr>
<td>ATC-2</td>
<td>195</td>
<td>405</td>
<td>200</td>
<td>390</td>
<td>228</td>
<td>4</td>
</tr>
<tr>
<td>ATC-3</td>
<td>285</td>
<td>390</td>
<td>260</td>
<td>338</td>
<td>345</td>
<td>16</td>
</tr>
<tr>
<td>ATC-4</td>
<td>205</td>
<td>80</td>
<td>180</td>
<td>110</td>
<td>92</td>
<td>9</td>
</tr>
<tr>
<td>ATC-5</td>
<td>120</td>
<td>90</td>
<td>150</td>
<td>154</td>
<td>169</td>
<td>8</td>
</tr>
<tr>
<td>Total:</td>
<td>17 hours 40 minutes</td>
<td>17.93 LOC hours 55 minutes</td>
<td>15 hours 899 LOC hours 389 minutes</td>
<td>18.00 LOC minutes</td>
<td>1046 LOC</td>
<td></td>
</tr>
</tbody>
</table>

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

a half hours in each tool. A three and a half hour execution time constitutes a gain of 78 percent compared to the execution time of the current manual test suite, 16 hours, if conducted by an experienced tester. Hence, automation would constitute not only an advantage in that it can be run automatically without human input but a considerable gain in total execution time which allows for more frequent testing. Potentially tests can run every night and over weekends and shorten feedback cycles in development. In Figure 3.4 the script development time for the scripts, taken from Table 3.4, have been visualized in a box-plot that shows the time dispersion, mean development time, etc. Using the mean development time, the development time for the entire automated test suite, all 50 test cases, can be estimated to approximately 21 business days for CommercialTool and 18 business days for Sikuli. The estimated development time for the automated test suite is in the same order of time that Saab spends on testing during one development cycle of the subsystem. Hence, the investment of automating the test suite is perceived to be cost beneficial after one development cycle of the subsystem.

The data in Table 3.4 was also subject to statistical tests to see if there was any statistical significant difference between the two tools. The data was
first analyzed with a Shapiro-Wilks test of the difference between the paired variables in Table 3.4, which showed that the data was normally distributed. Normal distribution allowed the data to be analyzed further with the Student t-test that had the p-value results 0.3472 for development time, 0.956 for execution time and 0.2815 for LOC. The Student t-test results were then verified with a non-parametric paired Wilcoxon test that had results with the same statistical implications. Hence, both the Student t- and Wilcoxon-tests showed that we cannot reject the null hypothesis, $H_0$, on a 0.05 confidence level. Therefore, it can be concluded that there is no statistical significant difference between the scripts of the studied tools in terms of development time, execution time or LOC. The statistical results are however limited by the few data points the tests were conducted on.

### 3.5 Discussion

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

Commercial Tool and Sikuli differ in terms of cost, vendor support, test functionality, script languages, etc., with impacts on different tool quality criteria, shown in Table 6.1, and are all important properties to consider for the industrial applicability of visual GUI testing. However, to show that visual GUI testing has any applicability at all in industry the most important aspect concerns the functionality of the image recognition algorithms.

The image recognition algorithms are what sets visual GUI testing apart from other GUI testing techniques, e.g. R&R, and also determine for what types of systems it is possible to apply the technique. R&R that interacts through GUI components was determined as unsuitable for the automation of the subsystem test cases since they had to interact with components not developed by Saab, e.g. interaction with custom and OS GUI components. These interactions required access to GUI component references that could not be acquired. The GUI components in the SUT, e.g. the simulators, windows in the OS, etc., did not always appear in the same place on the screen when launched. This behavior also ruled out R&R with coordinate interaction as an alternative for the study. Evaluation of visual GUI testing showed that it does not suffer from R&R’s limitations and therefore works in contexts where R&R cannot be applied. Visual GUI testing is applicable on different types of GUIs, evaluated in the pre-study experiments and in industry, which showed that both studied tools had high success-rates with non-animated GUIs and that Sikuli also had good success-rate on animated GUIs as well. Hence, this study shows that visual GUI testing works for tests on non-animated GUIs and perceivably also for animated GUIs. Non-animated GUI applicability is
The purpose of automation of manual tests is to make the regression testing more cost-efficient by increasing the execution speed and frequency and lower the required manual effort of executing the tests cases. Estimations based on the collected data show that a complete automatic test suite for the studied subsystem would execute in three and a half hours, which constitutes a 78 percent reduction compared to manual test execution with an experienced tester. Hence, the automated test suite could be run daily, eliminating the need for partial manual system tests, reduce cost, increase test frequency and lower the risk of slip through of faults. Mitigation of slip through of faults is however limited with this technique by the test scenarios since faulty functionality not covered by the test scripts would be overlooked, whilst a human tester could still detect them through visual inspection. Hence, the automated scripts cannot replace human testers and should rather be a complement to other test practices, such as manual free-testing. The benefit of visual GUI testing scripts compared to a human tester in terms of test execution is that the scripts are guaranteed to run according to the same sequence every time, whilst human testers are prone to take detours and make mistakes during testing, e.g. click on the wrong GUI object, etc., which can cause faults to slip through.

Scenario based system tests are very similar to acceptance tests and based on the results of this study it should therefore be concluded as plausible to automate acceptance tests with visual GUI testing. This conclusion is supported by the research of similar GUI testing techniques, e.g. R&R, which has been shown to work for acceptance test automation [23, 33]. Further support is provided by the fact that some of the manual test cases, categorized as fully scriptable, for the studied subsystem had been developed with customer specific data. The results of this study therefore provide initial support that visual GUI testing can be used for automated acceptance testing in industry.

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

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

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

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

### 3.6 Conclusion

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

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

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

Our analysis shows that visual GUI testing can overcome the obstacles of other GUI testing techniques, e.g. Record and Replay (R&R). R&R either requires access to the code in order to interact with the System Under Test (SUT) or is tied to specific physical placement of GUI components on the display. Visual GUI testing is more flexible, interacting with GUI bitmap components through image recognition, and robust to changes and unexpected behavior during testing of the SUT. Both of these advantages were important in the investigated subsystem since it had custom GUI components and GUI components that changed position between test executions. However, more work is needed to extend the tools with ways to specify and handle unexpected system events in a robust manner; the potential for this in the technique is not currently well supported in the available tools. For testing of safety-critical software systems there is also a concern that the automated tools are not able to find defects that are outside the scope of the test scenarios, such as safety defects. Thus any automated system testing will still have to be combined with manual system testing before delivery but the main concern for future research is the maintenance costs of the scripts as a system evolves.
Chapter 4

Paper C

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

E. Alégroth, R. Feldt, H. H. Olsson

ICST pp. 460-465
Abstract

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

To date, there are no industrial case studies, from the trenches, that visual GUI testing (VGT) works in industry when used by practitioners, nor data to support the long-term viability of the technique. In our previous work, we have shown that VGT is applicable in industry, even for testing of safety-critical software [37]. However, previous work has been essentially driven by researchers, e.g. they applied VGT techniques, compared the resulting test cases to earlier manual efforts, and then collected feedback and refinements from the industrial practitioners. There is a risk that this type of research does not consider all the complexities and problems seen by practitioners when actually applying a technique in practice. Furthermore, researcher driven studies are often smaller in scale and cannot evaluate longer term effects such as maintenance and refactoring of the test scripts or effects on, and of, changes to the system under test (SUT). Hence, there is still a gap in VGT’s body of knowledge regarding if the technique is applicable when performed by industrial practitioners in a real world development context.

In this paper we aim to bridge this gap by presenting an industrial case study from a successful project, driven entirely by industrial practitioners, with the goal to transition into VGT at the company Saab AB, subdivision security and defense solutions (SDS). The company chose VGT because of its ability to automate high system-level test cases, which previous automation techniques, e.g. unit testing [17, 18] and record and replay (R&R) [22–24], have had shortcomings in their ability to achieve. High system-level tests developed with automated unit tests have become both costly and complex, thereby spurring a discussion if the technique is applicable for anything but the low system-level testing, for which it was developed [20]. Furthermore, R&R techniques, which were developed for automation of system-level tests, are instead limited by being fragile to GUI layout and API change. Limitations that in the worst case have caused entire automated test suites to become inept [26]. Hence, the previous techniques have shortcomings in terms of flexibility, simplicity and robustness to make them long-term viable.

However, in this case study we show that VGT can overcome these limitations. Hence, showing that VGT has the capability to automate and perform industrial grade test cases that previously had to be performed manually, with equal or even greater fault finding ability, at lower cost. Capability provided by the technique’s use of image recognition that, in combination with scenario based scripts, allow VGT tools to interact with any graphical object shown on the computer monitor, i.e. allowing VGT scripts to emulate a human user. In addition, the study presents the practitioners’ views on using the technique, e.g. benefits, problems and limitations, when performed with the open source tool Sikuli [27]. Consequently, this work shows that VGT works for testing of real-world systems when performed by practitioners facing real-world challenges such as refactoring and maintenance of the SUT. The specific contributions of this work therefore include,

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

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

The concepts of using image recognition for GUI interaction is quite old and has been evaluated in a considerable body of knowledge. Work on using image recognition for GUI automation can be traced back to the early 90s, e.g. Potter [28] and his computer macro development tool, Triggers. Other early work in this area include the work of Zettlemoyer and Amant that used image recognition in their tool VisMap, which was used to automate the interaction with a visual scripting program as well as the game Solitaire [29]. However, this work focused on using image recognition for automation which we differentiate from testing since not all tools developed for GUI automation are intended for testing and vice versa.

The body of knowledge on using GUI interaction for testing is also considerable, e.g. shown by Adamoli et al. [22] in their paper on automated performance testing that covers 50 papers on automated GUI testing. Automated GUI testing can be performed with different techniques but the most common approach is referred to as record and replay (R&R) [22–24]. R&R consists of two steps. First a recording step where user input, e.g. mouse and keyboard interaction, to the system under test (SUT) is recorded in a script. In the second step, the recorded script can automatically be replayed for regression testing purposes. Different R&R tools record SUT interaction on different levels of GUI abstraction where the most common are on GUI bitmap level, i.e. using coordinates, or GUI widget level, i.e. using software references to buttons, textfields, etc. However, both approaches suffer from limitations that affect their robustness. Coordinate based R&R has the limitation that it is sensitive to GUI layout change whilst being robust to SUT code change. Widget based R&R, in contrast, is sensitive to SUT API or code structure change [26], but is instead robust to GUI layout change.

Image recognition based GUI testing with scenario based scripts, which we refer to as visual GUI testing (VGT), does not suffer from these limitations but it is only recently that the technique started to emerge in industry. One plausible explanation to this phenomenon is that the image recognition is performance intensive and it is not until now that the hardware has become powerful enough to cope with the performance requirements. VGT is a tool-supported technique, e.g. by Sikuli [27], EggPlant, etc., which conducts testing...
through the top GUI bitmap level of a SUT, i.e. the actual bitmap graphics shown to the human user on a computer monitor. Hence, scenario based VGT scripts can emulate a human user and can therefore also test all applications, regardless of implementation or platform, e.g. web, desktop, mobile. In most VGT tools the scenarios have to be developed manually, but there are also tools, e.g. JAutomate, which has record and replay functionality. Typical VGT scripts are executed by first providing the SUT with input, i.e. clicks or keyboard input, after which the new state of the system is observed, using image recognition, and compared to some expected output, followed by a new sequence of inputs, etc. In contrast to previous GUI testing techniques, VGT is impervious to GUI layout change, API or even code changes. However, VGT is instead sensitive to GUI graphics changes, e.g. changes in graphics size, shape or color.

Another approach to GUI testing is to use models, e.g. using finite state machines to generate test cases [30, 31]. These models generally have to be constructed manually, but automatic approaches, e.g. GUI ripping proposed by Memon [32], also exist. The benefit with GUI ripping is that it mitigates the extensive costs related to model creation. Costs that originate in the complexities of developing a suitable model. The limitation of this approach is that it is dependent on the SUT implementation, e.g. development language.

The area of GUI interaction based testing and automation is therefore quite broad but still limited in regards of empirical studies in real-world contexts with industrial grade software systems. R&R tools have been compared [22] and evaluated in industry, for both system- and acceptance-test automation, but, to our best knowledge, it is only our own work that evaluates VGT in an industrial context [37]. Our previous work is however limited since it was conducted only for a small set of real-world test cases and since the VGT automation was performed by researchers rather than practitioners. Hence, the body of knowledge on VGT, to the authors best knowledge, lacks industrial case studies that report on the real-world use of the technique.

Most research on GUI based testing focuses on system testing. However, acceptance testing is an equally important, valid and plausible test aspect to consider, i.e. tests where requirements conformity is validated through end user scenarios performed regularly on the SUT [13]. Scenario based acceptance tests do however distinguish themselves from system tests by including more end user specific interaction information, i.e. how the system will be used in the end users’ domain. Automated acceptance testing has also been a subject of much research, which has resulted in both frameworks and tools, including research into GUI interaction tools [33]. However, to the authors’ best knowledge, only our previous work has considered the subject of using VGT for acceptance testing.

4.3 Research methodology

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

Figure 4.1: Overview of the case study, including the two performed workshops and the continuous, yet discrete, communication between the company and the research team. Note that the academic support effort is considerably smaller than the VGT transition effort.

4.3.1 Research site

The case study presented in this paper was conducted in collaboration with, and at, the Swedish company Saab AB, subdivision SDS, in the continuation of this paper referred to as Saab. The study was conducted at the company because they had taken the initial steps towards transitioning into VGT to automate their current manual testing, which presented an opportunity to collect data to bridge the current gap regarding VGT’s real world applicability. Figure 4.1 visualizes the stages of the case study, which will be presented in more detail in the following section based on the guidelines for reporting case studies presented by Runeson and Höst, 2009 [1].

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

4.3.2 Research process

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

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

In the third stage of the study, which aimed to portray the project and its outcome, a second workshop was held on site at the company, during which two structured deep interviews were held with the driving testers, shown in the row named ‘Post-study’ in Figure 4.1. Additionally, at this point of the project, an additional tester had joined the transition project who could provide a new perspective and further information about the transition and usage of VGT. The purpose of the interviews was to verify previously collected data, get a deeper understanding of the transition project as well as to collect further data on challenges, limitations and solutions that had been identified. Both of the interviews were recorded and conducted using the same set of questions in order to raise the internal validity of the answers [1]. 71 questions were prepared for the interviews, 67 with the purpose of eliciting and validating previously collected information and 4 attitude questions aimed at capturing the testers views on VGT, post project. More specifically, the four questions were,

[A] Does VGT work? Yes/No, why?

[B] Is VGT an alternative or only a complement to manual testing?

[C] Which are the largest problems with VGT?
[D] What must be changed in the VGT tool, Sikuli, to make it more applicable?

In all of the questions, VGT refers to VGT performed with Sikuli [27], since Sikuli was the VGT tool that was used during the project. After the interviews, the recordings were transcribed in order to make the information more accessible. In addition, the answers were analyzed and compared among the respondents, i.e. the driving testers, to ensure that there were no inconsistencies in the factual data. The analysis showed that the respondents had answered the majority of the questions the same, including all attitude questions, but that they had complementing views on the attitude questions, i.e. what was the largest issue with working with VGT, etc.

4.4 Results and Analysis

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

4.4.1 Pre-transition

The VGT transition at Saab was initiated out of necessity to shorten the time spent on manual testing. For each release, every six months to one year, the SUT went through extensive regression testing where a selected subset of the SUT’s test cases were manually performed. Each regression test session had a budget of four to six weeks of man-hours. The test cases were documented in 40 test suites, referred to as acceptance test descriptions (ATD). Each ATD consisted of a considerable set of use cases (UC), e.g. roughly 100, which each defined valid SUT input and the expected output. On a meta level these UCs were linked together into test chains that defined the test case scenarios, as exemplified in Figure 5.6. A test case was defined as a test path through a test chain that could be either linear, or contain branches, where a set of UCs, UC1 and UC2 (Top left of Figure 5.6), were first executed to set up the SUT in a specific state. The set up was then followed by the execution of one of a set of optional UCs, UC3A-C (Middle of Figure 5.6) to create a test path. Test paths could also have varying length, as exemplified in the figure where UC3A (Middle left in Figure 5.6) is followed by UC3AA (Bottom of Figure 5.6) while the other two branches (UC3B and UC3C) lack following UCs. Hence, each test chain could contained a set of branching test-paths, i.e. test cases, defined by either common or unique UCs. The modular architecture of the manual test cases provided a lot of flexibility but was also considered tedious since some test chains required a lot of setup while only performing a small/short test thereafter.

The manual test period, four to six weeks, for the SUT, was then followed by a factory acceptance test (FAT) with the customer, executed over an additional two to three weeks, to validate the system, i.e. six to ten weeks of testing in total. However, a FAT would only be initiated if the manual tests had been executed successfully. Hence, transitioning to VGT from manual
testing would constitute a large gain for the company in terms of development time, cost and potentially raised quality, since a larger subset of test cases from the ATDs could be executed faster and at higher frequency [37]. Raising test frequency was also important since manual testing was the only means of testing the system, i.e. no other tests existed for regression testing purposes such as automated unit tests, etc.

Three VGT tools were evaluated for the project, i.e. EggPlant, Squish and Sikuli, to find one suitable for the VGT transition. A brief overview of the results of the evaluation is given in Table 4.1. The primary success factors during the evaluation, which took six man-weeks, were tool cost and script language ease of use. Each tool was evaluated based on its static properties as well as through ad hoc scripting and automation of actual use cases from the ATDs. In addition, the evaluation took into consideration the research teams’ previous work, i.e. comparison of different VGT tools [37].

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

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

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

![images/vgt_arch.pdf](images/vgt_arch.pdf)

Figure 4.3: VGT test suite architecture. TC - Test case, UCs - Use cases.

4.4.2 During transition

The VGT transition took place during roughly four calendar months, during which three representative ATDs were fully implemented into a VGT test suite. Representativeness was measured by the ATD's complexity, where two of the chosen ATDs were considered more complex than the average 40 ATDs, whilst the third was equal in complexity to the remaining ATDs. The VGT test suite architecture, visualized in Figure 5.3, consisted of two main parts. First, a main script for each ATD that imported all the automated ATD test cases, i.e. the test chains built from use cases. The second part was the test cases themselves which were executed by the main script according to the numerous test paths in each test chain. This architecture was required since Sikuli does not, as mentioned, provide any support for either development or management of test suites. The VGT test suite was also developed using external libraries, 'lib' in Figure 5.3. One of these libraries was a Python library for formatting and producing output. Output that could be viewed graphically through any web browser, i.e. the result of each test case was visualized as passed or failed in a table. Additionally, a Java library for taking screenshots was incorporated in the VGT test suite. The screenshots provided additional value to the result output by capturing the state of the system when a bug was identified, i.e. the faulty state of the GUI was captured for further analysis and for manual recreation of the bug. According to the testers, this functionality made it easier to explain, and present, the faults they encountered to the developers, thereby quickening SUT maintenance time. In addition, all global variables used in the scripts were placed in its own library called 'glob', whilst the 'cfg' library included all external paths, i.e. paths to where to save log files, find the external libraries, etc.
After automation of the three ATDs, the testers compared the VGT test suite’s execution time against the manual test suite execution time. Results showed an estimated speed up of a factor 16, from two work days (16 hours) to 1 hour for the two complex ATDs and from 1 day to 30 minutes for the third. Hence, the automation constituted a huge gain in test case execution time with no reported detrimental effects on bug finding ability, i.e. all bugs in the system that were identified using manual test practices could also be identified using the VGT test suite. In addition, due to the quicker execution speed, the automated ATDs could be run several times in sequence. The iterative test suite execution placed the SUT in states that the manual test cases did not cover. Consequently, three new faults were uncovered that had not been identified earlier with the manual testing. In addition, these bugs were automatically captured and recorded by the screenshot capabilities of the VGT test suite which made them simpler to present, recreate and motivate as faulty behavior to the developers. However, even with the much higher execution speed, the testers reported that they were often asked, “Doesn’t it execute quicker than this?” The simple answer, as reported by one of the testers, is, “Sikuli, or VGT, is limited by the speed of the SUT“, i.e. the VGT test suite cannot run faster than the reaction speed of the SUT’s GUI. Consequently, the scripts often had to be slowed down, using delays, in order to synchronize them with SUT loading times to ensure that the SUT’s GUI was ready for new input before the script continued its execution.

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

4.4.2.1 VGT test suite maintenance for improvement

To ensure validity of the automated test scripts, they were developed as a 1-to-1 mapping of the manual test cases, i.e. the manual tests were used as a specification for the automated scripts. However, later during the project, the VGT test suite was subject to maintenance. The maintenance done to the test case scenarios included, but was not restricted to, modification of the order of script operations, in order to provide smoother and quicker test case execution, and further modularization to facilitate strategic reuse. Hence, breaking the 1-to-1 mapping in some of the test cases. However, the purpose of each automated test case, i.e. the functionality the test case aimed to verify in the SUT, was kept the same. Consequently, a conclusion can be drawn that strict automation, i.e. 1-to-1 mapping, of the manual test specification may not necessarily be the best automation approach. Rather, the specification should only be used to specify what to test in the SUT, not necessarily how. The reason is because with automatic testing you can, and often want, to improve the test execution speed as much as possible, which can be done by
grouping certain actions together. In contrast, manual test scenarios need to be unambiguous and test actions defined logically to have high quality [38], which isn’t necessarily the fastest. Hence, the quality of a VGT script is greatly affected by how it is designed and implemented, i.e. narrowing the gap between testing and traditional software development.

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

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

4.4.2.2 VGT test suite maintenance required due to SUT change

Three calendar months into the VGT transition project a huge change was made to the SUT which included replacement of the map engine. Since the map engine was part of the core functionality of the SUT this change also affected the VGT test suite, i.e. causing 85-90 percent of the scripts to fail and thereby require some kind of maintenance, which included changing 5-30 percent of the images in every maintained script. The maintenance effort required to get the VGT test suite working completely again took roughly three man-weeks (240 man-hours) of work, which is to be related to the VGT test suite development time of three man-months (1032 man-hours). Hence, the estimated maintenance time of the entire VGT test suite, all 100 percent of the test scripts, would be 25.8 percent of the development time, i.e. 266 man-hours, which can be compared to the manual test budget of 480 man-hours per SUT development iteration. Note, the 4-6 week manual execution time, 120 hours, is with two testers. Consequently, the estimated development time of all 40 ATDs would be 13760 man-hours (7.6 man-years) and assuming
Table 4.2: Summary of development-, maintenance- and manual execution times (man-hours) and return on investment (ROI) (VGT test suite dev. time / manual exe. time) data acquired from the VGT transition project. mh - man-hour, h - hour

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

all of the tests broke, the maintenance time would be 3550 man-hours, equal to roughly 21 man-months of continuous work or equivalent to the budge of 7 iterations of manual testing, i.e. roughly 3.5 years. However, the time required to execute all of the 40 ATDs manually is estimated to 2400 hours. Hence, assuming that none of the tests required maintenance and the complete VGT test suite (40 ATDs) was executed continuously, i.e. 24 hours a day, the ROI for the entire development would be positive after roughly 8 days (199 hours), i.e. after executing all the 40 automated ATD’s 6 times. Additionally, for the three ATDs that were automated in the project, a positive ROI would be reached after 13 executions, i.e. after 32.5 hours of continuous execution, which is less than the time of the manual ATD execution, i.e. 80 man-hours.

These numbers, summarized in Table 5.7, do however not reflect the manual testing that is performed during the VGT test script development, required to validate test script conformance to the manual test specifications. Furthermore, the numbers do not take into account aspects such as the number of faults found during the test execution, i.e. quality gained from quicker feedback to the SUT developers and other benefits provided by the VGT test suite, e.g. identification of previously unknown bugs. With these aspects taken into account, the driving testers estimated that the currently achieved ROI of the VGT transition was neither positive or negative. Hence, their perception is that all future regression testing performed with the VGT test suite will provide positive ROI for the company. However, the numbers also show that it would be unfeasible to automate all the 40 ATDs since it would take 7.5 man-years. Hence, an important conclusion is therefore that a company may have to prioritize or be selective in which manual test suites they decide to automate. Furthermore, as described by the testers, VGT primarily solves cost and speed problems rather than raising quality. The higher test frequency can help identify bugs faster, but bugs are only found if covered by the test scenarios.
The testers encountered a set of additional problems during the VGT transition, which have been summarized in Table 4.3. The main problem was the volatility and instability of the VGT tool, i.e. Sikuli. Sikuli is still a release candidate, i.e. not a finished product, and therefore suffers from some lingering bugs. These bugs affect the stability of the tool’s IDE that is prone to failure in certain instances, e.g. if the execution thread of a script is manually terminated, or if the tool is terminated with an unsaved script, etc. The solution to solve these problems has been to only use Sikuli’s IDE for script development and instead run the developed VGT test suite from the command line, which was found to greatly improve stability.

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

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

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

4.4.3 Post-transition

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

Table 4.3: Summary of problems and solutions identified during the VGT transition project.
4.4. RESULTS AND ANALYSIS

testers driving the project. The purpose of the interviews was primarily to verify previously collected information but also to capture the testers views on if VGT is viable for system- and acceptance-testing in industry.

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

For the second question, when asked if VGT is a complement or a replacement for manual testing, the testers stated that it is a complement, “It’s part of the test palette”. Based on their perception, VGT may work as a replacement for smaller systems, but for large and complex systems it is neither suitable or plausible that this could be achieved. The reason is because it is improbable that test scenarios can be devised that cover all states of a large systems, which is equally unlikely for manual scenario based test cases. Instead, manual exploratory testing should be used to uncover new faults.

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

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

Due to the success of the transition project, i.e. identification of previously unknown faults in the SUT and the perceived cost-effectiveness of the technique, the use of VGT has also been accepted by the customer as a complement to the manual testing. Additionally, because of the success, the company plans to continue the automation of more ATDs and also develop a new VGT test suite to test all basic functionality of the SUT. This new VGT test suite will not be based on the manual ATDs but rather on domain knowledge about the intended low-level functionality of the SUT. The testers at Saab have also started looking at the possibility of creating an automatic thread-based
<table>
<thead>
<tr>
<th>Nr</th>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Does VGT work? Yes/No, why?</td>
<td>Yes, only technique the testers have identified capable of automating their manual tests.</td>
</tr>
<tr>
<td>2</td>
<td>Is VGT an alternative or only a complement to manual testing?</td>
<td>complement, since it can only find faults covered by the scripted scenarios.</td>
</tr>
<tr>
<td>3</td>
<td>Which is the largest problem with VGT?</td>
<td>The volatility of the tool and the image recognition.</td>
</tr>
<tr>
<td>4</td>
<td>What must be changed in the VGT tool, Sikuli, to make it more applicable?</td>
<td>Support for testing of distributed systems, e.g. through VNC.</td>
</tr>
</tbody>
</table>

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

exploratory test (TBET) based VGT application. TBET, a refinement of exploratory testing [39], is executed by following one or several execution threads, scenarios, through the SUT to find faults, and also their causes. However, no actual implementation had been conducted on such a solution at the time of the project.

Hence, it can be concluded that even though VGT has its limitations, challenges and problems, it is still a viable and applicable technique for industrial use when performed by practitioners. This conclusion is strengthened by the impact that the transition project has had within the Saab corporation where more Saab companies have started working with the technique. Even though, as reported by the testers, there are naysayers claiming that “Automation did not work 25 years ago and therefore it won’t work now.”. However, in this paper we have presented information that contradicts the naysayers claims, e.g. feasible development and maintenance costs, raised fault finding ability.

4.5 Discussion

The data collected during the industrial case study shows that the transition to VGT was both successful and of benefit to Saab, benefits summarized in Table 4.5. Firstly, the execution speed of the company’s previously manual tests was greatly improved that allows for greater test frequency and thereby faster feedback to the developers, i.e. from months to hours. Secondly, and perhaps more importantly, the automated tests did not just identify all the faults found by the manual tests, but also previously unknown faults. Consequently, this report provides support that VGT does not just lower testing costs, but can also helps raise software quality. However, as also reported, the transition cost of several large manual test suites can be extensive, so a
cost-benefit prioritization model of what test suites to automate should be developed, which is a subject of future work. Thirdly, the return on investment (ROI) of transitioning to the technique, i.e. automating the manual tests, was perceived by the driving testers to become positive after only one iteration of SUT development. A claim supported by our previous research [37], which came to the same conclusion at another Saab company. Additional support comes from the fact that the manual tests are continuously performed during VGT transition to ensure script validity, i.e. not taking time away from the normal manual testing, and the benefit of faster fault identification due to raised test frequency. Manual testing cost increases linearly with each development iteration, but VGT only has an initial cost for developing the automated test suites after which the cost of executing the scripts is constant. Hence, due to the execution speed of a VGT test suite, the number of executions required to reach a positive ROI can be performed quickly, as shown in Table 5.7.

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

Secondly, Table 4.5 presents data regarding the stability of VGT when used together with a virtual network connection (VNC). VNC was used during the project because the SUT was distributed over several computers. However, this pairing was recognized as a large problem since it lowered the success rate of the automated test suite, when it should have succeeded, to roughly 70 percent, i.e. due to image recognition failures. The VNC problem was identified by running a subset of test scripts, which could be run locally, against the SUT that resulted in a success rate of 100 percent, even when the tests were rerun continuously for 24 hours. Hence, Sikuli’s image recognition was not the source of the problem, but rather it was the third party VNC application, mitigated by local VGT test script execution. However, since the system was distributed over several computers, i.e. nodes, this solution instead limited which test cases could be executed. Hence, this was not identified as a benefit but rather an improvement of how to use Sikuli to raise test suite stability. Consequently, either a better VNC application has to be obtained or VNC should be integrated into Sikuli as already available in the VGT tool EggPlant, which was perceived by the testers be much more stable in this regard. However, EggPlant, as reported by the testers, had other limitations, e.g. a high cost and, what they considered, an unintuitive and more restricting scripting language. Consequently, existing VGT tools suffer from important, but different limitations, that makes it likely that manual test execution will still have to complement automated testing. However, the testers’ common view is that VGT both works and provides substantial value to the company,
<table>
<thead>
<tr>
<th>Description</th>
<th>Past</th>
<th>Current</th>
<th>Benefit (versus manual testing) or improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATD execution time</td>
<td>1-2 days per ATD (60 man-weeks for all ATDs), manually</td>
<td>0.5-1 hour per auto. ATD (Estimated 33 hours for all automated ATDs)</td>
<td>Test execution 16 times faster, higher test frequency, quicker feedback to developers</td>
</tr>
<tr>
<td>Fault finding</td>
<td>-</td>
<td>3, previously unknown, faults found</td>
<td>VGT provides greater fault identification ability, higher system quality</td>
</tr>
<tr>
<td>Test ROI</td>
<td>Linear cost (Manually)</td>
<td>Constant cost after 1 iteration (Automatic)</td>
<td>Positive ROI after one iteration, feasible development cost</td>
</tr>
<tr>
<td>Script maintenance</td>
<td>Unfeasible in the worst case for previous GUI test techniques</td>
<td>~25% of the development cost of the VGT test suite (Saab project, with Sikuli)</td>
<td>Maintenance cost perceived feasible</td>
</tr>
<tr>
<td>cost</td>
<td>(record and replay) [26]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sikuli executed over</td>
<td>~70% success rate with VNC</td>
<td>100% success rate without VNC</td>
<td>Sikuli stable when executed locally</td>
</tr>
</tbody>
</table>

Table 4.5: Summary of quantitative benefits and improvements identified during the VGT transition project.
even given the tools’ limitations.

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

4.5.1 Threats to validity

The main threat to the validity of this study is that it only presents results from one VGT transition project at one company. Hence, the results may have low external validity for other companies and domains [1]. In addition, since no structured data collection process could be performed by the driving testers during the project, due to resource constraints on the company’s end, quantitative metrics were only sparsely collected. The risk that very little quantitative data would be available from the project was identified already before the case study started and originates in the fact that this project was performed in a real-world context with real-world time constraints. Consequently, the results presented in this work are primarily based on data collected through interviews and are therefore mostly qualitative in nature. Further work is therefore required in more companies to provide additional support regarding the real-world applicability of VGT. Another threat is that the driving testers at the company might have been biased, i.e. wanting the transition to be successful. However, based on their thorough descriptions of faults, limitations and problems, this threat is considered minor.

4.6 Conclusion

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

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

Paper D

Transitioning System Tests to Automated Visual GUI Testing in Industry

E. Alégroth, R. Feldt, L. Ryrholm

EMSE pp. 460-465
Abstract

System- and acceptance-testing is currently mainly performed manually in industry. Test automation has been proposed as a general solution to all testing, but most automation techniques approach testing from a lower level of system abstraction, which make them unsuitable for high-level tests, i.e. system- and acceptance-tests. High system-level test automation techniques exist, e.g. capture/record and replay, but these techniques have limitations that hinder their long-term industrial use. However, a new automated test technique, referred to as Visual GUI Testing (VGT), has emerged, which combines scenario-based scripts with image recognition, which is perceived to mitigate these previous limitations. However, neither the industrial applicability of VGT, nor the challenges, problems and limitations (CPLs) of this technique, have yet to be determined. In this paper we present an empirical study, performed in two swedish companies developing safety-critical and mission-critical software, where the companies manual system test suites were transitioned into automated VGT test suites using the open source VGT tool Sikuli. During the transition, 58 CPLs were identified, which were grouped into 28 groups of mutually exclusive CPLs that we discuss in full. In addition, high-level, general, solutions to these CPLs are presented together with information about the VGT suites’ defect finding ability. Furthermore, metrics collected on the cost and return on investment of the VGT transition are reported. The conclusion drawn from the study is that even though there are many CPLs related to VGT, when performed with Sikuli, the technique still constitutes a valuable, flexible, cost-effective, technique for system- and acceptance-test automation.


5.1 Introduction

Current automation techniques for high-level testing, i.e. system- and acceptance-testing, leave more to be desired in terms of value, cost efficiency, and flexibility. This lack of automation support is one of the factors why companies persistently work with manual test practices, even though they are considered resource intensive, e.g. requiring developers or testers, customers, reference systems, are tedious, and are time consuming [24, 40]. These high system-level manual tests have been proven to be effective in finding defects on several levels of system abstraction, which has not been possible with previous automated techniques, e.g. automated unit tests [17, 18, 41]. Manual system- and acceptance-tests are thus key in the quality assurance of a system.

These test are generally scenario based, but one important distinction separates the two types of tests. Whilst system tests aim to verify the adherence of the software requirements, regardless if the requirements are valid or not. Acceptance tests, often performed by the customer or a customer proxy, are based on how the system will be used in the intended domain and therefore not only verifies but also validates the functionality of the tested system [13–15, 42]. Hence, it can be concluded that these types of tests are essential to ensure software quality, but that they are also costly. To make matters even worse, software is prone to change, which requires these tests to be rerun over and over, i.e. regression testing [43], raising costs even further.

To mitigate these costs, raise test frequency, and thereby software quality, automated testing has been proposed as the solution. However, most automated test techniques are intended for testing of lower levels of system abstraction, e.g. component testing with unit tests [17, 18, 41]. Attempts to use unit tests to perform system tests have shown that these tests become both complex and costly, both to develop and maintain [21]. These results have spurred the ongoing discussion if low-level test techniques, i.e. unit testing, with certainty, can be used effectively to test a system on a higher level of abstraction, i.e. system tests [20, 21]. Hence, unit testing does not fill the gap for an automated, high-level, valuable, cost effective, and flexible system- and acceptance-test technique. Attempts to bridge this gap has resulted in techniques such as Record/Capture and Replay (R&R) [22–24], with tools such as Selenium [44], WinRunner, etc. These tools use scripts that execute either against graphical components in the tested system or coordinates on the tested system’s graphical user interface (GUI). However, R&R has been shown to have limitations that still leaves more to be desired in terms of robustness to GUI layout and code change and maintenance cost. Consequently, R&R only partially covers the existing gap for an automated, high-level, valuable, cost effective and flexible technique for system- and acceptance-testing.

In our previous work [37], we performed initial evaluation of the industrial applicability of a tool-driven technique, referred to as Visual GUI Testing (VGT), which is emerging in industrial practice, and which has properties that transcend those of previous techniques. These properties originate from VGT’s use of image recognition that allows VGT tools to interact with the tested system in the same way a human user would, i.e. using the bitmap components of the system’s GUI. VGT’s industrial applicability was evaluated at the company Saab AB by comparing two VGT tools’, Sikuli [27] and a
commercial tool, which for confidentiality reasons shall be unnamed, ability to automate system test cases for a safety critical air traffic management system. 10 percent of the system’s manual test suite was automated, during which metrics were collected and analyzed. The tools were also compared based on their static properties. Results of the study showed that both tools were applicable in the industrial context and had almost the same functionality.

In this paper we provide further support for VGT’s industrial applicability through a case study of two VGT transition projects performed in industry. The two transition projects were performed in different companies within the Saab AB corporation and had the common goal to transition the companies current manual system test suites into automated VGT suites. The two projects were run in parallel, during which metrics and qualitative information were collected, from which challenges, problems and limitations (CPLs), for the transition and usage of VGT were extrapolated. CPLs that constitute the main contribution of this work. Thus, also contributing to the currently limited software engineering body of knowledge regarding CPLs related to automated testing [45]. Furthermore, we present general solutions to overcome, or mitigate, some of the identified CPLs. In addition, we present information regarding the defect finding ability of the VGT suites, costs of the VGT transition and the project’s evaluated return on investment. The specific contributions of this work thereby include:

- Identified limitations, challenges and problems that impact either the transition to, or usage of, VGT in industry.
- Identified practices that solve or mitigate CPLs that impact the transition to, or usage of, VGT in industry.
- Identified, detailed, information regarding the defect finding ability of the developed VGT suites in comparison with equivalent manual test suites.
- Detailed information on the cost of transition, and usage, of the VGT suites, contextualized in relation with the cost for manual test suite execution to evaluate return on investment.

In addition, these contributions are discussed and analyzed to answer the research question: *Can manual system test suites simply, flexibly and cost-effectively, be transitioned into Visual GUI Testing test suites, of value, for automated high system-level testing?*

The next section presents a background to manual testing and GUI testing as well as related work on previously used GUI based test techniques. Section 3 describes VGT in more detail, including a description of previous work. Section 4 describes the industrial case presented in this work. Sections 5 and 6 present the automated test suite solution and the robustness guidelines derived from the study. Finally Section 7 will conclude the paper and presents future work.

### 5.2 Background and Related work

The purpose of high-level testing, e.g. system- and acceptance-testing, is to verify and/or validate system conformance to a set of measurable objectives
for the system, i.e. the system’s requirements [42]. These tests are generally quite complex and therefore hard to automate, especially for testing non-functional/quality requirements such as performance, usability, etc. Furthermore, non-functional/quality requirements 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 of a system to be fulfilled, all 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, 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. scenarios of how the tested system will be used in its intended domain, which should be performed regularly, and preferably automatically, on the tested system [13–15, 42]. However, both manual scenario-based system- and acceptance-testing are costly, tedious and error-prone. Therefore, a considerable amount of research has been devoted to acceptance test automation, which has resulted in both frameworks and tools, including GUI interaction tools [23,33].

Additionally, the current body of knowledge includes a large amount of research into GUI interaction based automation, for both high-level functional and non-functional/quality requirements testing, as shown by Adamoli et al. [22]. In their work on automated performance testing, they identified 50 articles on automated GUI testing using different techniques. One technique that is commonly used is referred to as capture/record and replay (R&R) [22–24]. R&R is a two step approach, where user input, e.g. mouse and keyboard interaction, performed on the tested system are first recorded in a script. In the second step, these scripts are replayed to automatically perform the recorded scenario against the tested system for regression testing purposes. However, different R&R techniques record scripts 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 techniques approach testing by using exact coordinates on the monitor, i.e. where the GUI interactions was performed during the capture/recording phase. In contrast, R&R techniques that are performed on a GUI component level, also known as widget-based R&R, use a white-box approach, i.e. calling methods within the GUI components to interact with the tested system. However, both of these approaches suffer from different limitations that affect their robustness and usability but foremost their maintainability. The coordinate-based R&R techniques are sensitive to GUI layout change [24], e.g. displacement of buttons on the GUI, whilst being robust to changes in the code of the tested system. Widget based R&R techniques are instead sensitive to API or code structure change of the tested system [26], whilst being more robust to GUI layout change. However, the widget-based techniques, being white-box, i.e. requiring access to the backend of the system, are only applicable for tested systems written in one programming language and are mostly used to test web-applications.

Hence, previous techniques suffer from limitations that make them hard to apply on distributed systems, system of systems and cloud based systems.
However, in the emerging technique, referred to as Visual GUI Testing (VGT), which is based on image recognition in combination with scenario based scripts, these limitations of previous techniques are perceived to have been solved. VGT is a tool-driven technique, e.g. with tools such as Sikuli [27], which due to the image recognition can identify and interact with the graphical components on the top layer bitmap GUI, i.e. what is shown to the human user on the screen. 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, mobile, web. However, the body of knowledge on VGT is very limited and the technique is only sparsely used in industry, e.g. at Saab AB, Inceptive, Spotify. In our previous work [37], we performed an empirical, comparative, study with two VGT tools to identify initial support for the technique’s industrial applicability. The tools, Sikuli and a commercial VGT tool, which for confidentiality reasons shall remain unnamed, were compared on their static properties as well as their ability to automate industrial grade test cases for a safety-critical air traffic management system. Hence, the system used in Case 1 of the study described in this paper. In our previous work, 10 percent of the tested system’s 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 this automation. Thus, the study fulfilled the goal of providing initial support for the industrial applicability of VGT.

Automated testing is perceived by academics to be the solution to all test related problems, i.e. therefore all testing should be automated. However, as reported by Rafi et al. [45] 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 industrial 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 limitations of automated testing as well as the actual needs from industry. Hence, the need for lightweight, flexible tools with high learnability for simple, yet effective, test automation.

5.3 Industrial case study

The study presented in this report consists of two parts. First, a VGT transition project that builds on our previous work at Saab AB, in the swedish city of Gothenburg [37], which was driven by a researcher from the research team. Second, a VGT transition study in another Saab company within the
CHAPTER 5. PAPER D

Saab corporation, in the Swedish city of Järfälla, which was driven by industrial practitioners with minimal support by the research team. Hence, the two projects complemented each other by providing rigorous, academic information from the project driven by the researcher, and information about the real-world CPLs from the industrial practitioners in the other project. Thus, this paper presents the results from two holistic case studies [1] from two different companies, in the continuation of this report referred to as Case 1, the study in Gothenburg, and Case 2, the study in Järfälla. The two case studies were conducted in parallel, but with different research units of analysis [1]. Case 1 was driven by the research team where the unit of analysis was the VGT transition of one complete system test suite for a safety-critical air traffic management system, i.e. performed by experts from academia but with an industrial grade system. In particular, the focus of Case 1 was on the CPLs related to the transition and usage of VGT. Case 2, in contrast, was driven by industrial practitioners, with minimal support from the research team, where the unit of analysis was instead the success or failure of the project, which was performed in a real-world context with real-world constraints, e.g. 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 in the real-world.

5.3.1 The industrial projects

Both VGT transition projects were conducted in industry with two mature industrial software products/systems. In Case 1, the project was performed with a distributed, safety-critical, air traffic management system with an excess of 1 million lines of code, highly configurable to satisfy customer needs, developed in a multitude of programming languages. Additionally, the system was graphical user interface (GUI) driven with a very shallow GUI, meaning that most graphical components were continuously shown to the user, i.e. the graphical view of the GUI did not change during interaction. The system has been developed using both plan-driven and iterative development processes, starting with requirements acquisition activities, followed by development activities and finally testing activities. For each activity a set of artifacts were developed, e.g. design documents, user guides, but for this study the only document of interest was the manual system test description, developed and maintained through the course of a project for the system. The company’s developers were highly-educated with many years of industrial experience distributed between the different roles. Roles which are, product- and project managers and developers. Hence, the company does not have dedicated testers but instead use the developers, and their domain knowledge, to ensure the validity of the system. The company has a hierarchal, yet flexible, organization, that develop the studied system, distributed between two locations, i.e. Gothenburg and Växjö. In addition, the studied system was developed according to a quality assurance process that is compliant with the RTCA DO-278 quality standard. However, Case 1, due to resource constraints was only performed in Gothenburg. Furthermore, the tested system in Case 1 has several customers, consisting of both domestic, public and military, airports as well as international, public, airports.
5.3. INDUSTRIAL CASE STUDY

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

The system for which the VGT transition was performed in Case 2, is a battlefield control system which is distributed over several computers and is both safety- and mission-critical. However, due to the restricted involvement of the research team in Case 2, less can be revealed about the system’s details. The product is however mature, i.e. it has been developed, maintained and deployed for many years. Additionally, similar to the system in Case 1, the tested system in Case 2 is GUI-driven but with a GUI that has several views, i.e. the entire view of the GUI is changed during interaction with the system, e.g. when opening menus. Development of the system was, at the time of the study, performed using an iterative process with regular manual testing performed by dedicated testers. However, in contrast to the system developed in Case 1, the tested system in Case 2 did not follow any quality assurance standard. The graphical objects on the system’s GUI did however follow a military standard. Finally, the products main customer is the Swedish military contractor, Försvarsmaterialverk (FMV).

Figure 5.1 visualizes the research process and the three parallel tracks that were performed by the researchers in Case 1, practitioners in Case 2 as well as the support and data acquisition performed by the research team. The research support was continuous in Case 1, with the research team on site daily during the project. In Case 2, explicit support was only given during two full-day workshops on site, whilst all other communication was conducted during distinct instances over e-mail or telephone. However, in addition to providing support, the workshops performed in Case 2 were also the main source of data acquisition from the company, e.g. information such as what type of system they were working with, what the Sikuli test architecture looked like, etc. In the second workshop, two semi-structured, one hour, interviews were held, individually, with the industrial practitioners to make it possible to triangulate the acquired information. Triangulation was also performed against the information acquired earlier in the project. Consequently, the data acquisition in Case 2 was divided into three distinct phases, introduction (Workshop 1, in phase 1), data acquisition and implementation support through remote communication (Phase 2) and finally a retrospective analysis (Workshop 2, phase 3).

After the completion of Case 1 and Case 2, the collected data was analyzed, and the CPLs were extrapolated. In Case 1, the analysis was done through a combination of discussions with the leading researcher and document analysis. In Case 2, document analysis was used as well, but the primary source of information came from the conducted interviews, during which explicit ques-
tions regarding the CPLs, and solutions, were posed. The extrapolated CPLs were then categorized into three tiers, with 28 mutually exclusive CPLs on the lowest level of abstraction, i.e. Tier 3. Furthermore, the Tier 3 CPLs were analyzed to identify which were the most prominent CPLs in the projects, which resulted in eight generally applicable high-level CPLs. Prominence was evaluated based on occurrence in both projects, as well as more subjective measures, e.g. perceived negative impact of the CPL on the transition, or usage, of VGT, added frustration to the VGT transition, and perceived external validity of the CPL. In addition, potential solutions, or mitigation strategies, were identified and evaluated in a similar fashion, which resulted in four high-level generic solutions. Cost, and return on investment, was then evaluated based on the quantitative metrics that were collected from the two cases. Finally, conclusions were drawn from the analyzed data, which were validated by the leading researcher in Case 1 and the industrial practitioners in Case 2.

5.3.2 Detailed data collection in Case 1

The VGT transition in Case 1 started with an analysis of the automated test scripts that were developed in our previous work with Saab in Gothenburg. In parallel, an extended analysis was conducted of the manual test suite for the tested system in the project. This analysis was necessary since the tested system in the project was another version than the system used in our previous work. In addition, the analysis was required to mitigate a project threat related to leading researcher in the project who was unfamiliar with the system, inexperienced with VGT and lacked the details of our previous work. Furthermore, to reduce this threat further, and ensure quality of the collected information, an information collection process was put in place before the study started. The information collection process was based on a set of tables of metrics that were collected for each script and/or the test suite, e.g. development times, execution times, CPLs. Specifically, the information collection focused on sources and causes of CPLs that affected the VGT transition. Furthermore, the identified CPLs’ context were collected, e.g. in what script or aspect of the VGT tool, i.e. Sikuli, the CPLs were observed. This information was then used as the base for analysis for the results presented in this report.

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

Furthermore, like the system in previous work, it was distributed over three computers which required Sikuli to be paired with a third party Virtual Net-
work Connection (VNC) application in order to perform the test cases. The three computers of the test system were connected, through a local area network (LAN), to a fourth computer running a VNC viewer application and Sikuli. The test system setup has been visualized in Figure 5.2. Consequently, Sikuli, rather than executing scripts locally, executed the scripts through the VNC viewer application. Thus, remotely to the tested system to facilitate automation of test cases that required interaction with several computers within the distributed system.

Once the analysis of the manual test suite, and the setup of the test system had been completed, an architecture was defined for the automated VGT suite. The VGT suite consisted of a main script that imported and executed the individual test cases and help scripts including reusable methods. Furthermore, the VGT transition was performed by automating one manual test case at a time, since they were all mutually exclusive, according to a 1-to-1 mapping fashion between the manual and the VGT test cases. Hence, each test step of the automated test cases became directly traceable to the test steps of the manual tests. Additionally, to ensure test script validity and quality, the automated test cases were executed after each test step of the script had been completed, and verified against the results of corresponding manual test step. The automated test steps were defined in mutually exclusive methods, written in Sikuli script which is a scripting language based on Python, to make the scripts as modular as possible. Modularity was one of the keywords for the transition, applied in order to ensure reusability and maintainability of the scripts. Finally, after a test case had successfully been transitioned and executed against the tested system, it would be integrated into the VGT suite. In cases where erroneous test case behavior was identified post-integration, the test scripts were corrected and validated during execution of the entire VGT suite, i.e. execution of all the test cases in sequence. In addition, in order to ensure quality of the VGT suite and thereby the usability of the scripts for Saab, the scripts were subject to rigorous validation, which increased the development time of each script. This validation was performed by comparing the outcome of each test step with the outcome of corresponding test step in the manual tests, which was possible due to the 1-to-1 mapping between manual and VGT test cases.

The keywords for the automation in Case 1 were, as previously mentioned, modularity but also robustness. Robustness was achieved by implementing the scripts with three levels of redundancy for critical functions to mitigate catastrophic failure either due to image recognition failure, script failure, test system failure or detection of a defect in the tested system. In addition, all scripts were written as modular as possible to ensure reusability of generic functions and to ease script execution and validation. Modularization also
made it possible to run specific test steps out of order, thereby shortening script verification, since the entire test script did not have to be re-executed every time new functionality had been added. To further support verification and defect finding ability, a third party screen-capture software was built into the VGT suite. Hence, if a script failed, the tested system was reset to a known state by Sikuli after which the test scenario was rerun whilst being recorded by the screen-capture software. The recording functionality simplified script verification and was also used to identify defects in the tested system, i.e. if a script failed, a new video clip was recorded and saved which the developers could view in order to recreate the defect. The video-recordings were backed up by a textual log files that were created and saved for each executed test case.

5.3.3 Detailed data collection in Case 2

In contrast to Case 1, Case 2 was driven by practitioners and was started with a three week long evaluation of VGT. Three tools were evaluated during this period, i.e. Sikuli, eggPlant and Squish, to identify the most suitable tool to fulfill the company’s needs. After the evaluation, and because of recommendations from the research team, Sikuli was identified as the most suitable alternative for the automation. Similarly to the transition in Case 1, the industrial practitioners in Case 2 used their manual test cases as specifications for the automated test cases. However, since the testers possessed expert domain knowledge, not every test case was implemented as a 1-to-1 mapping to its manual equivalent test case. The deviation from the 1-to-1 mapping was required since the manual tests in Case 2, defined as use cases, were not mutually exclusive, but rather linked together into test chains that contained several test flows, where each test flow was a test case. 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. However, similar to the VGT suite developed in Case 1, the developed VGT suite in Case 2 was based around a main script that imported individual test cases and executed these according to a, by the user, predefined order. Furthermore, similar to Case 1, metrics and other information was collection during the VGT transition. However, in contrast to the systematic, and rigorous, information collection in Case 1, it was performed ad hoc by the industrial practitioners in Case 2, due to time and cost constrains. The collected information was then conveyed to the research team, as shown in Figure 5.1, through e-mail, telephone or during the interviews in Phase 3.
5.3 The VGT suite

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

The VGT suites developed in Case 1 and Case 2 were similar in terms of architecture and were both built in Sikuli script, which is based on Python. Sikuli has support for writing individual VGT based unit tests and includes special assertion methods for unit tests. However, Sikuli does not have support for creating test suites including several of these tests. Hence, in order to create a VGT test suite, of unit tests, using Sikuli’s supported functionality, all the tests have to be grouped into one large script, which has negative effects on reusability, maintainability, usability, etc, for large suites. Therefore, custom test suite solutions were created in both projects, i.e. in Case 1 and Case 2, using the Python scripting language support for object orientation and its ability to import scripts into other scripts. Hence, as mentioned, each VGT suite consisted of a main script that imported the individual test cases and executed these according to an order specified manually in a list in the main script. The architecture for the VGT suites is visualized in Figure 5.3. As can be seen from the figure, each script was given a setup method, a test method, containing the test steps of each test, and a teardown method. Consequently, the VGT suites were based on the test framework commonly used for automated unit tests, e.g. JUnit [41]. Additionally, user defined methods, variables to setup the VGT suite, etc, were extracted from the individual scripts and put in a set of support scripts that were then imported to the main script and/or the test scripts that required these methods. This extraction was performed to further modularize the VGT suite architecture. The modular, and hierarchal, architecture helped shorten development time, increase reusability and improve maintainability of the scripts. Hence, all common, and reusable, functionality was grouped in one location that made it easily accessible and modifiable during the VGT transition process.

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

In Case 2, the output was produced using an open source Python library that formatted the output from the test scripts into an HTML format, i.e. providing a graphical representation of the test results, similar to the output from automated unit tests, e.g. JUnit [41]. However, in contrast to Case 1, Case 2’s VGT suite did not record failed test scenarios, instead it only
took screenshots of the tested system’s faulty state when a test case failed, i.e. capturing the GUI’s faulty state when a fault occurred. The purpose for adding screenshots, and video recordings, to the VGT suites’ output was to provide the developers of the tested system with more information to simplify fault identification and recreation. This functionality also helped the testers to distinguish defects in the tested system from faults in the VGT suite itself, either caused by faulty test case implementation or image recognition failure during test script execution. Hence, mitigating the risk of false positives, i.e. reporting defects that were actually not defects.

5.4 Results and Analysis

During the VGT transition in Case 1, 58 challenges, problems and limitations (CPLs) were identified. These CPLs were categorized post-project completion as shown in Figure 5.4 into three tiers of VGT related CPLs. As show in the figure, a CPL related to Sikuli’s image recognition was classified as a Sikuli CPL whilst a defect in the tested system was classified as a test system CPL, etc. Consequently, three main categories of CPLs were identified, which, have been split into eight CPL sub categories based on their origin and/or root of cause, i.e. Tier 2 CPLs. These eight Tier 2 CPLs are: CPLs related to the version of the test system, the general test system, defects in the test system, CPLs related to the practices of the company, the tested system’s simulators, Sikuli, the test scripts or third party software. In Figure 5.4 these eight categories have been divided even further into a third level of abstraction with even finer granularity, with 28 CPL groups in total, i.e. Tier 3 CPLs. Hence, the figure shows the 28 mutually exclusive groups, not the 58 individually identified CPLs. The Tier 3 CPLs were mainly identified in Case 1 but corroborated by information provided by the practitioners in Case 2. However, not all CPLs could be corroborated, as shown in Figure 5.4, but are still considered as general CPLs that are likely, but not certain, to be present during any VGT transition project.

Out of the 58 identified CPLs, 14 were identified as Sikuli related. Five of the Sikuli CPLs were unique, i.e. mutually exclusive from any other CPL, eight were related to image recognition failure or tool volatility and the last CPL related to the VGT test suite that was developed. However, most of the CPLs were related to the test system. Out of the 58 identified CPLs, 20 were related to the version of the tested system, 20 related to the general test system and six were defects in the tested system, i.e. 32 in total. Hence, more than twice as many of the identified CPLs originated from the tested system compared to the tool, i.e. Sikuli. The remaining set of CPLs, i.e. 6 CPLs, were related to the third party software that was used in order to realize the VGT suite, e.g. the virtual network connection (VNC) application used to implement distributed manual test cases and the recording software, i.e. Camtasia.

As mentioned, the detailed information regarding the CPLs was identified in Case 1, but was corroborated by information acquired in Case 2. However, since the focus of Case 2 was not on capturing CPLs, but rather on the general success or failure of the VGT transition in an industrial context, when per-
Figure 5.4: A hierarchical tree diagram over the Challenges, Problems and Limitations (CPLs) that were identified in Case 1 and Case 2. The CPLs have been divided into three tiers of abstraction, with Tier 3 being the lowest level where the 58 identified CPLs have been grouped into 28 groups of mutually exclusive groups of CPLs. The model continues to the right (the grayed out symbols), connecting to Figure 5.5, which shows potential solutions to the Tier 3 CPLs.

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

formed by practitioners, information about CPLs were only sparsely collected. Therefore, information about the CPLs, in the continuation of this report, is primarily from Case 1, only supported by information from Case 2 if such information was available. However, it should be noted, as can be seen in Figure 5.4, that 18 out of the 28 mutually exclusive CPLs were identified in both cases, i.e. 48 percent of the CPLs. It should also be noted that these two cases were performed mutually exclusive of one another. Hence, strengthening the argument that these CPLs are generic for VGT, when performed with Sikuli.

### 5.4.1 Test system related CPLs

Test system related CPLs can be split into five sub-categories of CPLs, as shown in Figure 5.4. These CPLs are related to the version of the test system, the general test system, defects in the test system, specific to the company or the test system’s simulators. In this section, a summary of each CPL sub-category has been described including data related to the individual CPLs of each category. In addition, the CPLs affects on non-functional properties of VGT, e.g. usability, reusability, maintenance, have been described. Potential solutions to resolve or mitigate the CPLs have also been presented. However, since the focus of this paper is on the CPLs themselves, these descriptions are kept on a higher level of abstraction in terms of detail and will later be summarized in Section 5.4.5.

#### 5.4.1.1 Test system version

When dealing with complex systems, especially in the context of complex systems with several versions and/or variants, there are many different CPLs that can arise related to testing that also impact VGT. One such CPL, which was experienced in Case 1, was that the manual test specification was faulty, out of date or developed for another version or variant of the tested system, i.e. not aligned with the tested system. The cause of this CPL, in Case 1, was because the version of tested system, used in the project, was a build of the system, i.e. the product, which was used to demo the functionality of the product to potential customers. Thus, the tested system was limited in
terms of functionality, lacked documentation, etc. The reason why this system was used in Case 1 was because of resource constraints, i.e. the demo system required less hardware, only three computers compared to the product system which required 12, including three redundant servers. Consequently, since the VGT suite was implemented as a 1-to-1 mapping of the manual test cases, these limitations of this version of the tested system hindered how many of the manual test cases could be automated. In our previous work we performed automation with VGT on a version of the tested system that had 50 applicable manual test cases. However, for the product of the tested system, i.e. the full system, there are 67 manual system test cases. In Case 1 however, due to the limitations of the tested system in the project, only 33 out of the 67 test cases were applicable. Hence, restricting the usability, reusability and portability of the developed VGT test suite to other versions or variants of the tested system. However, all of the 33 manual test cases could, and were automated, in Case 1 and thereby constitute a complete automated system test suite for the tested system. Hence, providing Saab AB, in Gothenburg, with an automated system test suite for the tested system. Lack of functionality that prohibited automation of more manual test cases were, but not limited to, the system versions lack of roles, missing radar functionality, missing simulator support, etc. This functionality had purposely been omitted by the company when the tested system was developed to scale down the amount of required hardware.

Furthermore, several CPLs were identified in Case 1 that were related to the manual test cases for the tested system, e.g. some test cases were incorrect or out of date. Other tests were found to be ambiguous or aimed at testing functionality that was no longer part of any variant or version of the system. Hence, even though the tested system was a scaled down version of the product system the test cases were not applicable, in any version or variant of the system. These test cases were reported to the company and were considered a positive side effect of the automation since the faulty tests could be removed from the test specifications for several versions or variants of the system. Hence, from the product system as a whole. Thus, this CPL relates to, and shows, the complexity of keeping test specifications up to date as a mature system develops over time, i.e. test cases become obsolete as the functionality of the tested system evolves.

Additionally, several other CPLs related to the test specification were identified through manual analysis of the specification itself. However, most of these CPLs were uncovered during script development, since they caused the VGT scripts to find the same faulty system functionality over and over. During real-world execution of the VGT suite, i.e. during regression testing of the tested system, the execution of an individual VGT scripts would terminate after identifying a defect in the system and report the fault in the test output log before moving on to the next script. Hence, terminating the test script execution and rolling back the system to a known state after identifying a fault. Thus, a manual test case that identifies a fault at test step n can only be automated up to step n since the fault would prohibit the required interaction with the tested system defined by any step surpassing said step. Hence, all further execution of the test case, after identifying a defect, would be within an unknown, and potentially useless, system state making the interaction at test step n+1 invalid and/or useless. However, since the VGT suite, in Case 1,
was developed for future use in the company, for other versions and variants of the tested system, all test steps were still implemented. Thus, all test steps of the 33 applicable test cases were transitioned to VGT regardless if the test suite contained test steps that continuously found faults in the tested system. Thereby, raising the usability, reusability, and portability of the developed VGT suite, given that the faulty test step would pass for said other versions or variants of the tested system. However, to ensure that the test steps after a faulty test step, which were mutually exclusive from the faulty step, would complete, the assertion of the faulty step, e.g. test step n, was disabled in the test scripts to allow the test scenario to run its course. However, since these test cases ignored the faults, the continuation of the scripts, i.e. test steps n+1 and forward, potentially put the system in invalid states. Hence, performing interactions with the tested system that would not be valid in a real-world context. Six of these faults were identified, which will described later in the report. In a general system, a faulty state would not allow a scenario based test case to continue, due to a faulty system state. However, in the tested system this was possible because there were very few dependencies between the test steps, i.e. the test steps within the manual test cases were mutually exclusive. The mutual exclusion originated in the fact that the GUI of the tested system, in Case 1, was very shallow, i.e. all interaction components, e.g. buttons, were continuously shown on the tested system’s GUI. Hence, interaction in test step x, with x greater or equal to 1 but smaller than n being the last test step of a scenario, did generally not cause steps at x+1 to n to become unexecutable. Thus, causing the entire test case to be unexecutable.

Furthermore, in several test cases, the tested system’s faulty behavior was not consistent, i.e. not continuously detected during each execution of the VGT suite. In these cases the assertions were left active which gave these test scripts a certain probability of failure. Hence, the VGT suite in Case 1, provided the company, Saab AB in Gothenburg, with the ability to identify spontaneous faults that did not appear during every execution of the test suite, i.e. the manual test suite. Thus, providing the company with the possibility of identifying faults that would be very costly to identify through manual system testing.

Consequently, a conclusion that was drawn, during the transition to VGT in Case 1, was that VGT transition for systems early in their development cycle is more complex and lead to more partially implemented test scripts. However, the partially implemented test scripts can be justified since they can be of value for other versions and/or variants of the tested system. Additionally, a VGT transition earlier in the development cycle is hindered if there is a lack of functionality in the tested system, since VGT tests aim to test end to end functionality of the system, i.e. the system in its entirety, similar to system tests [46]. Hence, VGT suites, just as manual system- and acceptance-tests, provide better feedback and support later in the development cycle, since they require a larger degree of system maturity [46].

However, in contrast with manual system tests, VGT tests can be executed also earlier in the development cycle, due to the low execution cost. Furthermore, it is perceived to be more favorable to implement VGT suites later during the development cycle, or iteratively during development of a new system, i.e. adding VGT tests for new functionality as the functionality is
<table>
<thead>
<tr>
<th>CPL category</th>
<th>CPL sub-category</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test System</td>
<td>Test System version</td>
<td>20</td>
<td>0.344</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (General)</td>
<td>6</td>
<td>0.103</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (Defects)</td>
<td>6</td>
<td>0.103</td>
</tr>
<tr>
<td>Test System</td>
<td>Test Company specific</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (Environment)</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>37</td>
<td>0.5844</td>
</tr>
</tbody>
</table>

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

introduced [21, 47]. Hence, similar to the practices of unit test development, or the agile practice of test driven development (TDD) [48]. However, if the VGT transition is performed for a legacy system, a valid approach to the VGT transition is still to performed a big bang implementation, as it was performed in Case 1 and Case 2. We will return to the validity of this approach in Section 5.4.6, which discusses the return on investment of the VGT transition in Case 1 and Case 2. However, in order for either VGT transition approach, i.e. iterative or big bang, the script automation success is dependent on two factors. The first factor is that the automation is performed by a practitioner with domain and system knowledge, and the second factor that the tested system is part of an ongoing development project, i.e. a project that aims to add, change, or improve the functionality of a product. Both of these factors were fulfilled in Case 2, which perceivably added to its success. In contrast, neither of the factors were fulfilled in Case 1, since the demo version of the tested system was a one of, which was neither maintained nor being developed further within the company. Thus, perceivably adding complexity and additional CPLs to the VGT transition for that specific version of the system since it contained a series of faults and lacked functionality available in the product version of the system. From this discussion, two CPL mitigation practices can be concluded. First, that a manual test suite analysis should be performed preceding the VGT transition, by a domain expert with additional expertise of the tested system, to ensure the validity of the manual test suite. Second, that an iterative VGT script development approach, during system development, may be more beneficial in terms of return on investment and knowledge about faults found.

5.4.1.2 Test system (General)

One of the most time consuming CPLs that was identified, in both Case 1 and Case 2, was that Sikuli, due to the speed of Sikuli’s image recognition algorithm, is faster than the system’s GUI. Consequently, when Sikuli tries to perform the two subsequent interactions with the system’s GUI, the GUI
is not necessarily ready to receive new input, which can cause the script to fail. This CPL can be mitigated by adding delays in the scripts to synchronize the script’s execution, i.e. the interaction, with the system. However, this practice slows down the script execution time, especially since the delays either have to wait for the GUI to reach a stable state or include fixed delays based on the worst case execution time of the system. Hence, this practice, even though it adds robustness, usability, reusability and portability to the VGT suite, also adds maintenance costs, since these delays generally have to be maintained every time the system’s performance is changed. Thus, if the system’s performance is changed, the script has to be changed to lower the required time the script has to wait in order for the system’s GUI to become stable after an interaction. In addition, as reported by the practitioners in Case 2, this practice adds frustration to the VGT transition, especially for long scenarios since each synchronization, i.e. added delay, requires the script to be rerun from the start to verify correctness. Hence, if the added delay is not sufficient to synchronize the script with the system’s execution, the script has to be rerun from the beginning to ensure that the delays are long enough to complete the scenario. In addition, since the delays should preferably be as short as possible, but still ensure script robustness, this practice requires some trial and error which is both tedious and time consuming. To mitigate this problem, most VGT tools, including Sikuli, have special methods that delay the script execution until a sought image is found, i.e. the script is delayed until a stable GUI state has been reached. However, these methods require fine-tuning and therefore do not completely remove the synchronization CPL presented above. For web-based systems, this CPL is especially problematic since network latency has to be taken into consideration when adding the delays, i.e. sudden dips in network latency, if not mitigated in the script, can cause script failure. Hence, a conclusion can be drawn that the execution speed of a VGT script is governed completely by the execution speed of the tested system, i.e. the performance of the tested system’s state transition time. This conclusion is also supported by information collected in Case 2.

In addition, as reported by the practitioners in Case 2, a common comment by project managers and other developers viewing the automated script execution was: "Isn’t the execution faster than this?". This comment reflects, and reveals, an important perception about automated test techniques, i.e. the perception that all automated tests execute very fast. This perception is mostly true, e.g. automated unit tests can execute many hundreds, or even of thousands, of test case lines of code in minutes [47]. However, these automated test techniques, e.g. unit testing, approach testing from a lower level of system abstraction, e.g. through a white-box approach on a system component level. Hence, providing these techniques with lower usability, reusability and portability whilst raising the maintenance and modifiability costs for more complex, and/or high system-level, tests. In addition, most of these approaches do not provide interaction stimuli to the tested system in the same way as the end-user of the system. For instance, a lower level, white-box, unit test, generally only tests one or a couple of system components at once, thereby disregarding the timing constraints between components in the system, as a whole, which will appear during actual usage. Attempts have been made to use tests of lower level of system abstraction, e.g. unit tests, for system test automation.
However, these tests have quickly become very large and complex and therefore unmaintainable [20, 21]. VGT scripts take these timing constraints into consideration by emulating end user interaction, but, as mentioned, at the cost of test execution speed. However, based on information from both Case 1 and Case 2, VGT tests can still execute up to 16 times faster than their manual equivalent tests. Unfortunately, since the scripts are bound to the execution speed of the tested system, it is unlikely that greater speedups, than the ones reported by this study, can be reached, even for faster systems. However, no further analysis has been performed to validate this claim. Hence, VGT scripts execute considerably slower than unit tests, which may be a CPL for industrial VGT adoption since the general perception of industrial practitioners, as mentioned, is that automated testing is extremely fast. However, what is then failed to be considered is the higher level of system abstraction of these tests, which add to the usability, reusability and portability of the tests.

Yet another identified test system related CPL originates from the developer’s/tester’s lack of domain knowledge and ambiguity in the manual test descriptions/specifications. Hence, if the developer/tester performing the VGT transition also lacks domain knowledge, it may be impossible for him/her to implement a specific test or he/she might implement it incorrectly, whilst a domain expert would have been able to resolve the CPL. This CPL was observed in Case 1, as already mentioned in 5.4.1.1, where the test specification that was used for the test automation was intended for another variant of the tested system, causing a mismatch between the test specification and the tested system. Hence, many of the test cases could only partially be implemented, or not be implemented at all, because the tested system lacked functionality. Hence, lowering the usability, reusability and portability of the implemented test scripts. To resolve the CPL the test specification that was initially used in Case 1 was replaced with another version of the same specification. However, it was soon found that the new version of the test specification included other test cases that could not be implemented on the version of the tested system used in the project, i.e. causing new CPLs. The origin of these CPLs was the tested system itself, which as mentioned, was a demo version of the product for which there had not been either resources, e.g. time, to develop a correct test specification. Hence, the faulty specification that was provided, which was intended for another variant of the tested system, was still the best possible test specification available. Thus, limiting the VGT transition, as previously discussed, in terms of which tests that could be automated, etc.

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

Hence, verifying faults in the tested system through several redundant failure mitigation schemes during test execution. It should be noted that defects within the tested system, caused by faulty system behavior, were identified with equal certainty as the manual, equivalent, test cases, i.e. 100 percent of the time.

5.4.1.3 Test system (Defects)

Much of the discussion about faults and defects was covered in section 5.4.1.1. However, Case 1 also uncovered defects that were previously unknown or only partially known to the company. Hence, these defects had not been discovered previously and not corrected in later versions or variants of the tested system. The reason why these previously unknown defects were uncovered was due to a combination of two factors. First, due to the thorough analysis, which included manual execution of manual test cases, required for the leading researcher performing the VGT transition in Case 1, to understand how the test cases was supposed to be executed. Second, due to the quicker and more cost effective execution of the VGT suite compared to manual tests. The second factor was the one with the highest impact on the defect finding ability since some of the identified defects were not consistent, i.e. could not always be replicated, neither manually or automatically. For instance, some of the tab menus in the system sometimes did not load properly. However, since there is no cost related to running the VGT suite it could be run several times in a row and thereby find these faults. In combination with the video recording functionality of the VGT suite, in Case 1, the faulty behavior of the system could be determined and the defects could be reported in more detail to the company. Thus, the recordings could served as a means to identify the cause of the faults. Hence, showing how the video recording functionality of the VGT suite adds to its robustness and
usability. The defects that were uncovered were of different nature, from faulty GUI functionality, e.g. tabs not launching, to complex faults in the services of the tested system’s backend, e.g. missing alarms intended to warn the system user of incorrect or missing input from external interfaces. However, even though some of the defects could be considered minor, the tested system was safety-critical and therefore all faults or defects were considered high risk. Hence, the identified defects and faults were all of value to the development company.

Similar information regarding the identification of faults and defects was acquired from Case 2, where the speed and low cost of executing the VGT suite made it possible to identify three previously unknown defects in the tested system. However, in Case 2 the cause to why the defects had previously not been identified differed from the cause in Case 1. Previous to the automation, in Case 2, the test cases were always executed in a linear order, i.e. starting with test case 1 to test case n, and only executed once every development iteration, i.e. the test suites were executed once every six months. However, the automated tests, which were equivalent to these manual tests, could be executed several times every week. In addition, as reported by the industrial practitioners in Case 2, these tests could be executed several times in a row without additional cost. Hence, by doing so the tests put the tested system in previously untested, yet valid, states and thereby uncovered the faults that were previously unknown to the company. Consequently, the practitioners in Case 2 reported that if it hadn’t been for the automated tests these faults had never been found. In addition, this information shows that because the test case roll backs were not implemented correctly, the system state was slightly altered between the executions and thereby changed the tested systems initial state, i.e. the base state of the tested system was different between the test suite executions. In turn, this information indicates that the order of test case execution, i.e. the tested systems state, can be vital to uncovering faults. Hence, depending on what state the system is in when the test case starts, a faulty system state may or may not be found. Thus, providing support for our previous discussion, i.e. since there is no cost associated with running the VGT suite, it can help uncover new defects by being executed several times in a row. Furthermore, due to the low cost, the test scripts can be executed in different, even random, order, which on a test case meta-level, i.e. suit level, should trigger different states in the tested system, thereby covering more states of the tested system that could potentially contain faults. Thus, perceptively, increase their effectiveness [43]. Additionally, since tests are generally performed, in industry, under time and cost constraints it is not possible to execute several different sequences of the same manual test cases. Hence, VGT suites, provided that they are implemented with a good architecture, allow several different sequences of individual test cases to be performed almost without cost. Thus, covering more system states and sequences of interaction that may appear during real-world usage of the system. However, it should be noted that some of these sequences may be invalid, i.e. never appearing in real-world use, but potentially still add value to the development company by showing what features of the system, in what scenarios, cause faulty system behavior. Consequently, the capability of VGT suites to be executed several times in different sequences, without additional cost, add to their usability.
5.4. RESULTS AND ANALYSIS

5.4.1.4 Test company specific CPLs

As mentioned, there was a mismatch both in terms of the version and variant of the tested system in Case 1 and the test specification that was used for the VGT automation. Consequently, since the researcher driving the automation lacked the domain knowledge to defer this mismatch, many of the test cases could only partially be implemented, if they could be implemented at all. In addition, due to this mismatch, many of the implemented test cases also required maintenance once the mismatch was uncovered, i.e. the test scripts required refactoring. Further analysis why the wrong specification was provided by the company, for the tested system, discovered that the version control of the system’s documentation was in some cases under par and prone to mistakes. Mistakes such as incorrect test specifications being used to test a version or variant of the system. This analysis was performed through discussion with employees, at the company where the Case 1 was conducted, who stated that it happened from time to time that the wrong test specifications was used during testing. However, this had never caused any larger problems since the company’s developers possessed domain knowledge which made it possible for them to detect the inconsistencies between the test specification and the tested system rather quickly. Consequently, it can be determined that version and variant control of large complex systems with several projects being developed at once is a non-trivial CPL that, potentially, can lead to costly mistakes. Mistakes that in Case 1 added to the VGT suite development cost since a faulty manual test specification, for the tested system, was used during the VGT transition. However, this CPL is not VGT specific, i.e. it impacts regular manual system testing as well, since it’s related to the complexity of product management, but it still has an impact on the VGT transition process [49]. Hence, this CPL adds further support that any VGT transition project should be lead by a domain and/or test system expert since the faulty VGT transition, due to the use of a faulty manual test specification, can reduce the usability, reusability, portability and maintainability of the developed VGT suite.

5.4.1.5 Test system (Environment)

Another CPL that was identified during Case 1 was related to the tested system’s simulators. The intended use of the tested system is to control the landing lights and radar equipment at an airport. However, since the actual hardware equipment, e.g. a radar station, is not available during development and testing at the company, the company instead uses a set of different simulators to stimulate the tested system’s external interfaces. Hence, simulating the hardware interfaces using software which is a common practice in industry [37,50]. These simulators are maintained by the company and as the tested system has evolved, so has the simulators, i.e. new, or additional, functionality has been added to the simulators to better test the system. Additionally, new simulators have been constructed over time to test new functionality of the tested system that the previous simulators were not able to test, e.g. timing constraints on the input data which is sent over the external interfaces. However, not all simulators are compatible with all versions or variants of the tested system. Thus, limiting what test cases that could be automated for the
5.4.2 Test tool related CPLs

The CPLs discussed so far have all been related to the tested system and are thereby potentially generic CPLs that can affect any VGT transition, with any VGT tool. However, in both Case 1 and Case 2, the VGT tool Sikuli was used, and the following section will present specific CPLs identified for that tool. Two sub-categories of CPLs were identified as Sikuli specific, i.e. CPLs related to the tool and CPLs related to the VGT suite that was developed.

5.4.2.1 Test tool (Sikuli) related CPLs

Sikuli is a tool developed and maintained in an open source project managed by the Michigan Institute of Technology (MIT). Thus, the tool is still under development and is currently, at the time of writing this report, still a release candidate, i.e. not a finished product. Consequently, the tool is volatile, containing several defects that affect the stability of the tool’s image recognition algorithm, its behavior in general and its integrated development environment (IDE). The defects related to the tool’s IDE cause it to crash and freeze sporadically, forcing the user to restart the tool, which, as reported from the leading researcher in Case 1 and the industrial practitioners in Case 2, adds to the frustration of working with the tool. Thus lowering its robustness and usability. However, as also reported by the practitioners in Case 2, the stability of the tool can be improved by running the scripts from the command-line version of the tested system in Case 1, requiring older simulators to be used in the automated scripts. The use of older simulators during the automation will therefore require the developed VGT suite, in Case 1, to be maintained for future use in other versions or variants of the tested system. Hence, restricting the usability, reusability and portability of the developed VGT suite. The required costs of this maintenance is however unknown and therefore a subject of future work, i.e. an analysis of the required cost of migrating the VGT suite from one simulator environment to another. However, based on the research teams’ expert assessment, none of the interactions with the new simulators were determined to be unimplementable. Hence, even though the migration to a new simulator environment may be related to a considerable cost, there is no evidence to suggest that it would be impossible. The basis for this expert conclusion is because all simulators at the company use standard components, e.g. buttons, from the Windows operating system which Sikuli, through empirical tests, has shown to be fully compatible with.

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

<table>
<thead>
<tr>
<th>CPL category</th>
<th>CPL sub-category</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Tool</td>
<td>Test tool (Sikuli)</td>
<td>13</td>
<td>0.224</td>
</tr>
<tr>
<td>Test Tool</td>
<td>Test scripts</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>14</td>
<td>0.2412</td>
</tr>
</tbody>
</table>
rather than the tool’s IDE. No explicit reason why this practice raised the robustness of the scripts, but it was perceived, by the industrial practitioners, to be related to defects in the tool’s script engine.

During test script execution, it is often required to terminate the execution manually, e.g. due to recognized faulty behavior. However, manual termination of the script execution can cause the scripts to be corrupted. Hence, a serious CPL, which was identified both in Case 1 and Case 2, with impact on robustness and usability of the tool. In the instances where the CPL occurred, it was reported that the script logic could always be recovered but that the links to the sought images in the script all broke, which required recapturing of all the images. Hence, for larger scripts, this CPL can be both time consuming and tedious whilst also adding frustration. The mitigation practice found for this CPL was once again to run the scripts from the command-line rather than from Sikuli’s IDE. Thus, restricting the IDE’s use to script development. Additionally, this CPL has been identified on both Windows and MacOs versions of the tool, but with perceivably higher frequency on Windows.

The most prominent Sikuli related CPLs are however connected to the tool’s image recognition algorithm, e.g. it randomly fails in cases where it has previously succeeded, randomly clicks on generic positions next to the sought images. These random failures were identified in both Case1 and Case2 to be the source of the most frustration during the VGT transition since the source of the problem has yet to be identified and appears completely random in nature. Image recognition failure often occurs during script development because the similarity level of the sought image is either too high or too low for the image recognition algorithm to find a match, e.g. because there are several similar images on the screen. To ease the script development in these instances, Sikuli’s IDE has a built in feature that allows the user to preview what the image recognition algorithm considers a match to the sought out image on the tested systems GUI. A screenshot of the widget within Sikuli’s IDE that supports this feature is shown in Figure (Figure removed, requires text edit). As can be seen in the screenshot, the image recognition algorithm has found a match, indicated with a bright red square on the target application. If several matches had been found, these had been ranked according to similarity of the found image compared to the sought image. The ranking is visually displayed with squares of different color, from light blue, for a match of low similarity, to bright red, for a match of high similarity. During script execution, Sikuli will always interact with the image with the highest similarity. According to the leading researcher in Case 1 and the practitioners in Case 2, corroborated by the authors’ own experiences, this feature has never failed to find a match, i.e. indicating a 100 percent success rate of the image recognition algorithm. However, it was reported that even though the preview feature indicates that a match, of high similarity, can been found in the tested system’s GUI, this does not ensure that the image recognition will find the sought image during script execution. Furthermore, even if the link to the sought image has been corrupted in the script, as described above, this feature still shows a match. The successful matching of corrupted images added a lot of confusion and frustration in both Case 1 and Case 2, since this feature’s behavior did not align with the behavior of the script, i.e. the widget found a match but the script failed. Consequently, this feature does not guarantee image recognition
success even though it successfully indicates a match. Thus, having a negative impact on the robustness, usability, reusability and maintainability of Sikuli scripts.

Another CPL connected to this functionality is that the image recognition algorithm is often too lenient on what is considered a match to the sought out image. Therefore, a lot of effort during script development is required to fine-tune, i.e. raise or lower, the similarity level of individual images, which in its default setting is set to 70 percent of the original image, as indicated by the bar in the lower right of Figure ???. For larger scripts, in particular, this practice becomes quite time consuming, especially since no pattern has been identified as to which images require higher, or lower, similarity level than the default setting. Furthermore, as mentioned above, Sikuli’s scripts sometimes get corrupted, requiring images to be recaptured. Recaptured images do, however, not retain the image similarity properties they had before they got corrupted. Hence, the similarity level once again needs to be set manually. Consequently adding further to the frustration to the script development but also lowers the usability, portability and maintainability of the scripts. Furthermore, the need to perform this practice is perceived to be one of the main contributors to maintenance costs. However, no empirical study has been performed to verify this claim and is therefore a subject of future work.

Other CPLs related to the image recognition, identified in Case1, concerns inconsistent ability of finding certain GUI graphics, i.e. animated images and images with similar appearance but with slightly different color. The identification of animated images, e.g. blinking or moving images, was required since the tested system in Case1 had animated buttons that indicated alarms or warnings to the user. For instance, in most airports there are nets that can be raised in order to stop an aircraft in an emergency, e.g. during brake failure. These nets are situated on opposite ends of the runway and should be raised dependent on from which direction the aircraft is approaching the runway. However, in case the wrong net is raised, or both nets are raised at the same time, the GUI warns the user by periodically switching the background color of the buttons, used to raise the nets, from red to yellow and back again in an interval of roughly one second. Testing this functionality, with confidence, showed to be problematic for Sikuli that often failed to find the buttons, depending on which state they were in. A conclusion can therefore be drawn that Sikuli is limited in its capability to interact with animated GUIs, i.e. lowering the tools usability. However, as mentioned, the tested systems, in both Case 1 and Case 2, were both quite static, i.e. only included minor animations. Thus, a larger study with a completely animated GUI is a subject of future work to verify the conclusion stated above.

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

Yet another CPL with Sikuli, experienced in both Case1 and Case2, concerns the lack of documentation about Sikuli’s scripting language. Sikuli’s scripting language, called Sikuli script, is based on Python but has been extended with a set of methods that make use of the image recognition capabilities of the tool. However, as reported by the leading researcher in Case 1 and the industrial practitioners in Case 2, there is no consistent, searchable, API for all of the methods supported by Sikuli script and what these methods’ properties are. As an example, Sikuli script has a method called wait, which originates in Python’s sleep function, which delays the script execution for t number of seconds. The method can take a series of different input parameters, i.e. wait(image, t), wait(t, image), wait(image) and wait(t). These input alternatives are not specified in the official Sikuli documentation but all have different behavior. The first alternative, wait(image, t), causes Sikuli to pause the execution for a maximum of t seconds or until it finds the sought image “image”. However, if the second alternative, wait(t, image), is used, the script will always pause for t seconds and then try to find the image “image”. Both functions are useful in different circumstances, but since the user generally takes the worst case scenario into account when setting the wait time, the second alternative will make the script execution time much longer. This was considered a large CPL in both Case 1 and Case 2 until it was discovered that the wait method had these different capabilities. In addition, this CPL, i.e. the lack of proper documentation for Sikuli script, can discard companies from using the tool. Especially since some of Sikuli script’s methods are unintuitive, e.g., the wait method. Hence, this CPL lowers the usability, learnability, reusability, portability and maintainability of the scripts. However, as we identified in our previous work [37], Sikuli script, or Python, is in general an intuitive programming language, even for novice users with limited programming experience.

Both the tested systems, in Case 1 and Case 2, included input areas that accepted swedish words as input. Hence, there were manual test cases that required the user to input swedish words and/or letters, i.e. å, ä, ö. However, Sikuli only supports an english keyboard for typing, i.e. the supported method “type” does not support swedish letters. Thus, in order to use the swedish letters, Sikuli’s “paste” method must be used instead, but this method does not work in all instances since it uses the operating systems (OS) clipboard which isn’t available, for instance when typing passwords into Windows OS login screen. The solution to this CPL, used by the leading researcher in
Case 1, was to type ascii characters as combinations of pressing the ALT key followed by the ascii code of the letter. Identifying the solution was both time consuming and added frustration to the research member. In addition, the fact that Sikuli does not support characters from other languages was puzzling since the tool’s IDE can be set to a variety of different languages, including swedish. Consequently, Sikuli scripts have limited usability, and portability to systems with GUI’s in languages other than english.

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

Consequently, there are many serious CPLs related to the Sikuli tool, some of which are related to the tool’s IDE, others to the image recognition algorithm, etc. The cause of many of these CPLs, but not all, has been determined to be because of the tool’s Java implementation that includes native methods, i.e. it is not platform independent. Due to these native methods we have found, through exploratory experimentation on a set of roughly 50 different computers, that the stability of Sikuli is highly dependent on what version of the Java Runtime Environment (JRE) is installed. In addition, the current version of Sikuli, at the time of writing this report, requires Java 6, i.e. JRE 6, and is only stable for certain versions of said JRE. However, no comprehensive evaluation has been performed to find which versions of JRE 6 that make Sikuli more stable, but is a subject of future work. In addition, for users of Java 7, Sikuli’s initialization file has to be modified to use the exact path to the JRE 6 executable, rather than the path provided by the operating system’s environmental variables. Thus, many users, especially users with limited programming and/or OS knowledge, can be discouraged from using the tool, if they get it to work at all. In addition, our exploratory experiments could also show that Sikuli is more stable, in general, on MacOs than on Windows. Consequently, this CPL limits the robustness, usability, reusability, portability and maintainability of the scripts and use of the tool.

5.4.2.2 Test application

The VGT suites that were developed in Case 1 and Case 2 were both developed as 1-to-1 mappings of the manual test cases. However, the manual tests were, in both cases, designed to be as unambiguous and simple as possible for a human tester to perform. Hence, the test cases were defined in mutually exclusive steps where some configuration was made of the system, through manual input, to put the system in a specific state before the assertion, i.e. the test, was performed, followed by another setup, input, etc. Thus, the tester could start with test step 1 in test case x and then, without requiring any knowledge of future steps, execute each step to test step n. The benefit of this approach is that any tester, or even developer, can perform these tests. The drawback is that they take more time since the tester, potentially, has to jump back and forth between, for instance, a simulator and the tested system, i.e. set up the simulator, do the test, do a new setup, etc. However, many test steps, at least in Case 1, were mutually exclusive, meaning that all the setup
of the simulator could have been done in one step rather than several. Thus, grouping all of these test steps had made the test case more complex and could potentially have added ambiguity to the test. However, when executing a test case automatically, ambiguity in the script is seldom a concern due to the more structured semantics of a script compared to natural language. Rather, there are other aspects that are more important such as script performance, quality and reusability of the scripts. Hence, a 1-to-1 mapping between manual tests and VGT scripts is not necessarily the best approach. On the one hand, the 1-to-1 mapping approach allows the scripts to be verified through comparison with manual test execution, but on the other it can have negative effects on the performance of the script. An alternative automation approach is therefore to group all related interactions with similar GUI component in one place, given that they are mutually exclusive and do not affect the flow of the test scenario. The benefit is that the execution time becomes lower, whilst making it possible to verify the test case outcome with the manual scripts. The CPL lies in identifying these mutually exclusive test steps and group them together in the script. Furthermore, this practice contains a trade-off, since it raise maintenance costs since changes to the manual test cases, that also have to be implemented in the test scripts, become harder to implement.

Another potential automation approach is to disregarding the manual tests all together. Hence, instead of using the manual tests as a specification, instead use domain expertise to build an automated test suite for the core functionality of the system. The drawback of this approach is that it requires domain and system experts to write the scripts, which might be a CPL due to the associated cost. A similar approach is to only automize the test suite’s large and complex test cases, e.g. test cases that are prone to faulty execution by a human, or test cases that are so long that they become cumbersome for a human to execute. These alternative approaches are supported by information acquired from the practitioners in Case 2 that in the future will focus on developing a more generic VGT application to test all the basic functionality of the tested system, i.e. not following the manual test specification.

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

An alternative solution to the text marking CPL, discussed above, which was found to be more robust, is to search for text using different applications search functions, e.g. in text editors. The search function was used in Case
<table>
<thead>
<tr>
<th>CPL category</th>
<th>CPL sub-category</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>Third party</td>
<td>10</td>
<td>0.172</td>
</tr>
<tr>
<td>software</td>
<td>software</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>10</td>
<td>0.172</td>
</tr>
</tbody>
</table>

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

1 in test cases that required XML files to be rewritten in order to change the layout of the tested system’s GUI. The drawback of this approach is that you have to know the text you are looking for and it only works in systems that have a search function. The third alternative is therefore to use Sikuli’s Optical Character Recognition (OCR) algorithm that, as mentioned, allows the tool to transform text in images to strings that can be used for further processing. However, Sikuli’s OCR algorithm is unreliable in the available version of the tool, at the time this report was written, and a recommendation is to use it as sparsely as possible.

5.4.3 Support software related CPLs

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

However, the use of VNC was also the cause of several CPLs. First, when Sikuli is executed locally it can remove the mouse-pointer from the screen, making the mouse-pointers location irrelevant for the success of an image recognition. However, when Sikuli is executed over the VNC application, the mouse pointer cannot be removed, since it is rendered remotely on the target computer. The mouse pointer can therefore obstruct buttons, and other sought images, during image recognition, thereby causing the image recognition to fail. Additionally, the execution over VNC also lowered the success rate of the image recognition due to network latency. Even when the VNC application was running in an optimized setup, the frame rate caused the image recognition to fail. This CPL was especially troublesome for test cases that included
5.4. RESULTS AND ANALYSIS

<table>
<thead>
<tr>
<th>Tier 1 CPL category</th>
<th>Tier 2 CPL category</th>
<th>Nr. of occurrences</th>
<th>Percentage of all CPLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test System</td>
<td>Test System version</td>
<td>20</td>
<td>0.344</td>
</tr>
<tr>
<td>Test Tool</td>
<td>Test tool (Sikuli)</td>
<td>13</td>
<td>0.224</td>
</tr>
<tr>
<td>Support software</td>
<td>Third party software</td>
<td>10</td>
<td>0.172</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (General)</td>
<td>6</td>
<td>0.103</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (Defects)</td>
<td>6</td>
<td>0.103</td>
</tr>
<tr>
<td>Test System</td>
<td>Company specific</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td>Test System</td>
<td>Test System (Environment)</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td>Test Tool</td>
<td>Test scripts</td>
<td>1</td>
<td>0.0172</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>58</strong></td>
<td>~1 (0.998)</td>
</tr>
</tbody>
</table>

Table 5.4: Summary of distribution of problems, challenges and limitations (CPL) that were identified during the automation process ordered according to occurrence of the Tier 2 CPLs. Tier 1 CPL-categories have been listed for each sub-category.

interaction with animated graphical components, e.g. the emergency nets in the tested system in Case 1, since the low frame-rate could cause the buttons to be distorted as they were toggling color. Hence, on occasion, the buttons were rendered with the top half of the button in red color, and the button half in yellow color, or vice versa. However, other, less obvious, faulty behavior was also introduced. The increased faulty behavior was observed in both Case 1 and Case 2, but VNC was first identified as the source of the CPLs in Case 2. Consequently, running Sikuli remotely increases the usability of the tool, i.e. test cases for distributed systems can be executed. However, in the same time it lowers the robustness of the tool, since it increases the chance of image recognition failure.

In addition, as found in Case 1, the choice of VNC application is a relevant factor. At the start of the project, a more simplistic VNC application was used, but it was soon discarded due to poor performance and because it was unable to send keyboard commands to the tested system. These keyboard commands, e.g. CTRL+ALT+DELETE and CTRL+V, where used to simplify, and/or where required, to perform some of the test cases. Hence, a new VNC application had to be acquired. The second application solved the CPL regarding the keyboard commands and also increased the stability and speed of the remote image transfer. However, it did not solve these CPLs completely, forcing the leading researcher to add of more failure mitigation redundancy in the scripts.

5.4.4 CPL Summary

58 challenges, problems and limitations (CPLs) where identified during this project, primarily through analysis of the information collected in Case 1, at Saab in Gothenburg, corroborated by information provided from Case 2, i.e.
<table>
<thead>
<tr>
<th>Nr.</th>
<th>Title</th>
<th>Description</th>
<th>Q-attr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Image recognition volatility</td>
<td>The image recognition algorithm randomly fails for reasons unknown causing failures in scripts that previously worked. This problem is assumed to be related to Sikuli not being a finished product.</td>
<td>Rob, Usa.</td>
</tr>
<tr>
<td>2</td>
<td>Image recognition limitations</td>
<td>The image recognition algorithm has limited ability in finding certain animated objects and differentiating between objects with similar color, e.g. dark to light gray.</td>
<td>Rob, Usa.</td>
</tr>
<tr>
<td>3</td>
<td>Negative VNC effects</td>
<td>Running Sikuli through a VNC application allows testing of distributed systems, however it also increases the chance of image recognition failure due to VNC updates, color changes in the GUI, mouse cursor placement, etc.</td>
<td>Rob, Usa, Port.</td>
</tr>
<tr>
<td>4</td>
<td>Sikuli IDE volatility</td>
<td>Sikuli is not a finished product and therefore has faults that cause it to crash, loose links to images in scripts, fail to start, etc. Note that these CPLs do not include image recognition related CPLs.</td>
<td>Rob, Usa, Port, Reu, Main, Mod.</td>
</tr>
<tr>
<td>5</td>
<td>1-to-1 mapping</td>
<td>Using manual test cases as the specification for the automated scripts is not always feasible nor appropriate in terms of performance, maintainability, etc. Furthermore this documentation can be faulty.</td>
<td>Usa, Port, Reu, Main.</td>
</tr>
<tr>
<td>6</td>
<td>Test system limitations</td>
<td>Test system CPLs appear on a case by case basis and are hard to generalize. A reoccurring CPL is however that the tested system is usually slower than the script execution, e.g. requiring delay statements to synchronize the scripts and the tested system. This is especially true for web systems.</td>
<td>Rob, Usa, Reu, Port, Main.</td>
</tr>
<tr>
<td>7</td>
<td>Hardware limitations</td>
<td>Image recognition is a performance heavy operation and lack of hardware support therefore influences the success rate of the image recognition, especially in real-time systems with animated interfaces.</td>
<td>Rob, Usa, Reu, Port.</td>
</tr>
<tr>
<td>8</td>
<td>Test system maturity</td>
<td>The scripts require the images of the interaction components of the tested system’s GUI. These can be captured from the tested system directly given that all functionality of the test has already been implemented. Faulty behavior of components will be found during automation but also hinders completion of the script scenarios.</td>
<td>Rob, Usa, Reu, Port, Main.</td>
</tr>
</tbody>
</table>

Table 5.5: A summary of the eight most prominent challenges, problems and limitations (CPLs) encountered during the projects, i.e. Case 1 and Case 2. These CPLs were chosen based on their prominence during the projects, measured based on occurrence, perceived negative impact on the transition to, or usage of, VGT, added frustration, etc. **Rob** - Robustness, **Usa** - Usability, **Reu** - Reusability, **Learn** - Learnability, **Port** - Portability, **Main** - Maintainability.
Saab in Järfälla. The analysis made it possible to break down the CPLs into three tiers, with the lowest, Tier 3, containing 28 mutually exclusive groups of CPLs, as shown in Figure 5.4. Table 5.4 summarizes, numerically, how these 28 groups where divided over the Tier 2 CPL categories, and in turn how the Tier 2 CPLs were divided over the Tier 1 CPLs, which there are only three. The top three CPLs concern either the tested system itself, the test tool or support software not directly connected to the tested system itself, e.g. the third party virtual network control (VNC) software. Analysis of the collected CPLs shows that most of them relate to the tested system itself, rather than the testing tool, i.e. Sikuli. However, the most prominent CPL concerned the testing tool’s image recognition algorithm, which sometimes failed without reasonable cause, had limited ability to interact with animated graphical GUI components, etc. In Table 5.5, the eight most prominent, out of the 58 collected, CPLs have been summarized, together with their impact on quality aspects, such as usability, robustness, etc., of VGT, performed with Sikuli, on industrial systems similar to those developed in Case 1 and Case 2. These eight CPLs were chosen based on occurrence during the project, but also more subjective measures, e.g. frustration, which was elicited during interviews with the leading researcher in Case 1 and the industrial practitioners in Case 2. Furthermore, these eight CPLs have been described as generic as possible, i.e. on a higher level of abstraction, such that they can be more objectively compared in other industrial contexts, e.g. for other types of systems.

5.4.5 Potential CPL solutions

The focus of this article are the CPLs related to VGT, performed with Sikuli. However, during the project several potential solutions were found to solve, or mitigate the identified CPLs, of which four generic solutions have been summarized in Table 5.6 together with the CPLs, from Table 5.5, that they solve or mitigate. These four generic solutions were extrapolated from the ad hoc solutions that were used during the project in either, or both, Case 1 and/or Case 2. The reason for the low number of solutions, in this report, is because many of the solutions that were found in the project were ad hoc and thereby not generalizable. In Figure 5.5 the four generic solutions have been mapped to the Tier 3 CPLs, i.e. continuing Figure 5.4, which is shaded gray to the left in the figure. As can be seen from Figure 5.5, there are still many CPLs connected to “other or no solution”, i.e. representing CPLs that were either solved or mitigated with ad hoc practices, or could not be solved at all. In addition, Table 5.6, and Figure 5.5, shows how these generic solutions are applicable for mitigation of several of the most prominent CPLs. It should be noted that these solutions were primarily extrapolated from information acquired in Case 1, but corroborated with information from Case 2, since the information from Case 2 was not as detailed regarding CPLs and solutions, or mitigation of these CPLs as Case 1.

The first generic solution, which was used during the project, was to ensure that the scripts had redundant exception handling to mitigate CPLs such as tool and image recognition volatility, etc. Redundant exception handling was achieved by using exception handling, provided by Python, to capture image not found exceptions. Developing this exception handling is however quite time
CHAPTER 5. PAPER D

consuming and complex since it is most often caused by unexpected tool or system behavior, which cannot be anticipated. However, through smart script development, the failure mitigation can be built into the test suite architecture, as performed in Case 1. The failure mitigation in Case 1 consisted of three levels of failure redundancy, i.e. on a method level, on a script level and on a test suite level. Hence, if a method failed in a test script, the test suite would try to rerun the test step. If the rerun failed, the system would roll back the entire script to put the system in a known state and then rerun the entire script. If the test case failed again, on the same step, the test suite would restart the tested system to ensure that it was in a clean, and known state, and then rerun the test script once more. For each rerun a textual log of the test execution was automatically produced and for every rerun above the test step level, the execution was also recorded. The failure mitigation in Case 2 was less complex, with only one level of redundancy. Hence, if a test case failed, the system would roll back the system, and then continue with the next test case. Rollbacks were in both cases performed with a teardown method similar to the JUnit test framework [41].

The second solution, to mitigate the lack of Sikuli documentation, is to continuously, during development of the automated scripts, document the architecture, e.g. model the test suite architecture, document the functionality of help scripts. Thus, ensuring that new testers, and/or developers, can more easily start working with the test suite, if they are required to do so, but more importantly to mitigate degradation of the VGT suite architecture [21]. This solution shows, one of many ways of, how VGT testing has a lot in common with traditional software development. However, documentation is only required for the test architecture, since the scenario-based scripts are intuitive by themselves. In addition, given that the scripts are implemented as 1-to-1 mappings of the manual test cases, the manual test case descriptions also serve as specifications, and documentation, of the test scripts. Consequently, documentation of the automated tests is required, but this need is restricted to the architecture itself as well as reusable help scripts and methods.

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

The fourth solution aims to solve the CPL that VGT scripts generally execute quicker than the tested system can update its GUI, i.e. synchronization discrepancies between the scripts and the tested system. This CPL was reported, in both Case 1 and Case 2, to be a huge source of frustration during the automation and was in addition reported to be very time consuming. However, as also reported in Case 1 and Case 2, this CPL could be mitigated through systematic insertion of delays in the scripts at locations which could, later during the project, be determined upfront through experience of working with the tool. In Case 1, this solution was also supported by the development of custom methods with an additional time delay parameter. For instance, the click(img) method from Sikuli’s instruction set, where “img” would be the sought image in the tested system’s GUI, was expanded to create a click(img, delay) method which waited “delay” number of seconds before performing the click. These custom methods provided additional robustness to the scripts but also increased execution time since these methods required the sought image to be found twice, first by Sikuli’s wait for image function and second by the click function, i.e. doubling the minimum number of required image recognition sweeps. However, due to the increased robustness, the leading researcher in Case 1 reported that it was still beneficial, especially since the image recognition algorithm in Sikuli is quite fast, i.e. can perform upwards of 5 complete image recognition sweeps of the computer monitor per second.

Finally, even though several generic solutions and mitigation practices were found for the CPLs, there were still CPLs that required ad hoc solutions. These solutions were specific to this project and could therefore not be generalized. In addition, there were some CPLs that could not be solved or mitigated, e.g. faulty manual test specifications, which required larger effort to be solved, e.g. changes to the development company’s documentation process, and where therefore out of scope for this project. Hence, unsolvable CPLs can be grouped into two categories. First, CPLs for which potential solutions could be identified, but which required so much effort that they were out of scope for the project. Second, CPLs that no solution was found at all, e.g. how to mitigate Sikuli’s volatility. However, these unsolvable CPLs appeared very seldom, some only once during the entire project, based on information from Case 1. This information was partially corroborated by information from Case 2, where the industrial practitioners, as mentioned, had identified the tool’s CPLs but that these CPLs were, in relation to the benefits of the technique, manageable.
<table>
<thead>
<tr>
<th>Nr.</th>
<th>Title</th>
<th>Description</th>
<th>Solves problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Redundant script failure mitigation</td>
<td>Adding failure mitigation code in the scripts, i.e. exception handling, on several levels of the script can help mitigate tool and image recognition volatility as well as unexpected system behavior. 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.</td>
<td>Image recognition volatility, Image recognition limitations, Negative VNC effects, test system limitations.</td>
</tr>
<tr>
<td>2</td>
<td>VGT script documentation and user defined methods</td>
<td>In order to raise script robustness, usability, reusability, etc., it is recommended to construct user defined methods and other reusable artifacts. However, in order to preserve the knowledge of these artifacts, they should be thoroughly documented to make it easier for new developers to start working with Sikuli script. In addition, this documentation should include descriptions of native Sikuli methods to complement the official, but lacking, Sikuli documentation.</td>
<td>Sikuli IDE volatility, 1-to-1 mapping.</td>
</tr>
<tr>
<td>3</td>
<td>Local/Remote script execution</td>
<td>Removing the remote system control application, e.g. virtual network control (VNC) application, from the VGT suite makes it possible to raise the stability of the script execution. However, this also limits what test cases can be executed for distributed systems. Another drawback is that the image recognition is performance intensive and can therefore modify the behavior of the tested system. Hence, in some cases the use of VNC is required but its use should be limited if possible.</td>
<td>Negative VNC effects, 1-to-1 mapping.</td>
</tr>
<tr>
<td>4</td>
<td>Systematic SUT synchronization</td>
<td>The VGT suite often executes quicker than the tested system reacts to input and therefore requires the scripts to be delayed. By creating custom methods that take timing input and smart usage of the ”wait for image method” in Sikuli script, a process can be constructed to iteratively synchronize the script with the tested system.</td>
<td>1-to-1 mapping, test system limitations.</td>
</tr>
<tr>
<td>5</td>
<td>Other or no solution</td>
<td>For several of the CPLs ad hoc solutions had to be formulated. These solutions included, but were not limited to, analysis and replacement of the manual test suites, hardware and software reboots to reset the tested system and/or Sikuli, change of VNC application, development of partially implemented test cases, etc. For the sake of completeness, we also state here that there were CPLs that could not be solved, e.g. the missing system functionality, stopping Sikuli from getting corrupted, etc. Hence, there are CPLs that should be investigated in future work to be solved and/or mitigated.</td>
<td>Image recognition volatility, Image recognition limitations, Negative VNC effects, Sikuli IDE volatility, test system limitations, Hardware limitations and test system maturity.</td>
</tr>
</tbody>
</table>

Table 5.6: A summary of general solutions that were identified for the CPLs listed in Table 5.5.
5.4.6 Defect finding ability, development cost and return on investment (ROI)

Thus far, this report has focused on the CPLs of implementing or using VGT for high system level test automation, and the number of CPLs have been considerable. However, the industrial practitioners in Case 2, which was performed in a real-world context under real-world time constraints, still consider VGT to be a valuable and cost-effective technique. They reported that they did not encounter any functionality that they could not automate using VGT, only that it was more or less challenging. However, since VGT is a test technique, the primary measure of success is still the defect finding ability of the developed VGT suites. For the tested system in Case 1, six different defects were identified.

[A] Switching between military and civil landing lights did not work.

[B] The button to switch military and civil landing lights did not disable as intended.

[C] Switching quickly between runways caused the tested system to freeze.

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

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

[F] The button to switch military and civil landing lights did not disappear as intended.

These defects were identified during implementation and execution of the VGT suite and were all reported as identifiable with the manual test cases as well. In addition, it was reported that no further defects were found with the manual test cases. Hence, the VGT scripts were able to identify all the defects that the manual tests could identify, i.e. providing the VGT suite with the same level of confidence, in terms of defect finding ability, as the manual tests.

In Case 2, the industrial practitioners reported similar results, i.e. that their automated tests could identify all the defects that the manual tests could identify. Additionally, as mentioned, they reported that the VGT suite could identify three defects that previously had been unknown to the company. In the past, the manual test cases had only been run once in sequence every development iteration, i.e. starting with test case 1 and ending with test case n. Hence, the same sequence every time. However, by running the VGT suite several times in a row they altered the internal state of the tested system thereby testing new, but by the practitioners perceived valid, states that were not previously tested. These states were not considered unlikely to appear during real-world use of the system but were simply not covered by any manual test case scenario. Hence, these results show the importance of what order the individual test cases are executed. Furthermore, since the execution order of automated test cases is simple to change, VGT allows the user to quickly and cost-effectively cover more meta-level scenarios, at almost no additional cost. However, for this practice to work, the test cases need to be mutually exclusive, such that they can be run out of order. In addition, it is important
Table 5.7 summarizes some of the cost metrics that were acquired in Case 1 and Case 2. Worth mentioning, again, is that Case 1 was performed with only one researcher whilst Case 2 was performed by two industrial practitioners. In addition, the researcher in Case 1 had limited development and testing knowledge whilst the practitioners in Case 2 were domain experts. Hence, there were several aspects that may have affected the results from the two cases. Additionally, only one manual test suite was automated in Case 1, whilst in Case 2 a total of three suites were automated. Two of the test suites in Case 2 were considered to be more complex than the average test suites used at the company whilst the third was considered equal in complexity compared to the average. In addition, the manual test suite that was automated in Case 1 was considered to be of roughly equal complexity as the more complex test suites in Case 2. Furthermore, since the structure of the manual test suites differed quite extensively between Case 1 and Case 2, the complexity comparison is still a subject of discussion. In some aspects, the manual test suite in Case 1 was more simplistic, whilst in other regards the suite used in Case 2 was simpler. 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 loosely coupled use cases. These use cases, examples shown in the right of Figure 5.6, were then tied together on a meta-level to form test scenarios, as shown in the left of Figure 5.6. Figure 5.6, on the left, exemplifies a test chain used in Case 2 which is made up from three different test flows, or scenarios. These scenarios all start with use case 1 and 2, i.e. UC 1 and UC 2, and are then followed by one out of three exchangeable use cases, i.e. UC 3A, 3B or 3C (Middle of Figure 5.6). However, the manual test structure also allowed test scenarios to be of unequal length, exemplified with UC 3AA, bottom left of Figure 5.6. The three different test chains are then joined again, and completed, by UC 4 (Bottom of Figure 5.6). This structure perceivingly improved the manual test cases usability, maintainability and reusability since
5.4. RESULTS AND ANALYSIS

Figure 5.6: 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.

Figure 5.7: Boxplot showing the execution times for individual test scripts from the VGT suite developed in Case 1. Time, shown on the y-axis, is measured in minutes.

the use cases could simply be switched out in the middle of a chain to test newly added functionality of the system. The drawback of this approach, for manual testing, is that many of the test scenarios become very similar and therefore tedious to test, i.e. the tester has to perform the same interactions over and over. However, once automated, the tediousness no longer becomes a problem but the benefits, e.g. reusability and maintainability, are kept intact, since new scripts can easily be formed by reusing the individual use cases together with a newly developed scripts.

Table 5.7 shows the development time of the VGT suites in Case 1 and Case 2. As can be seen from the table, the development time in Case 1 was considerably lower than the development time in Case 2. However, the time in Case 1 is based on very exact measurements, performed by the leading researcher, and only consists of the actual time spent on script development. In contrast, the development time measured in Case 2 was performed by industrial practitioners in a real-world context where measuring the exact development time was low priority compared to the development itself. Consequently, the measured time from Case 2 contains more overhead than the measured time in Case 1. In addition, more emphasis was put in Case 1 on making the test suite as robust, reusable and flexible as possible, i.e. adding more redundancy and focus on the script architecture than in Case 2. This additional development increased the overall development time in comparison to Case 2 where the focus was on developing the fastest solution possible. Hence, the manual test architectures, expertise of the script developers, and time measurement methods, all differed between the two cases. Hence, the data in Table 5.7 concerning development time are from two different contexts and should therefore be treated as such, i.e. not compared directly without taking the context, and measurement methods, into account.

However, Table 5.7 does show some interesting, and comparable, differences in terms of execution time speed-up between the manual suites and the developed VGT suites. In Case 2, the speed-up was quite considerable, i.e. by a factor of 16, whilst in Case 1 the speed-up was only marginal, i.e. 1.06. The reason for this gap is relates to the individual test cases, and what they aim to test. Hence, the manual test suite used in Case 1 contains several
quite large test cases, test cases that contain loops and tests to test the safety requirements of the system. The looped test cases aimed to verify the functionality of all the buttons of the GUI, i.e. the same test scenario applied to every button, grouped together into one test case to save space. Performing these tests manually is very time consuming, and tedious, and are therefore often skipped, i.e. only a few buttons are tested during test suite execution. However, with the VGT script all of the buttons can be verified during each execution, providing better coverage, without additional cost. However, these tests, even when performed automatically, take quite some time to execute due to the sheer number of buttons that are tested. Hence, some of the VGT scripts had considerably longer test execution time than others, visualized in a boxplot in Figure 5.7. The boxplot shows that the average execution time of the VGT scripts was roughly 27 minutes, with a standard deviation of approximately 44. Thus, showing huge deviations in the execution times of individual scripts. The cause of this deviation is visualized Figure 5.7, which shows that there were several outliers with much higher execution time, i.e. at most approximately 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 is capable of determining for which buttons it is necessary to repeat all the steps of these looped test cases. For a junior tester, the manual test suite can take upwards of 40 hours, as presented in our previous work [37]. 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 execution time of the automated scripts can also be contributed to the fact that several of the manual test cases requires 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. If the connection is not established within that time, an alarm is triggered. 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 have to wait. These tests have significant impact on the total execution time, 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 automated in Case 2 were more on a functional level and did not consider timing issues, etc, which made it possible for the VGT suite to gain a larger performance advantage over the manual test cases.

However, if the above statement is true, how can there be any speedup of the execution time at all? One factor that explains the speedup of the VGT scripts, compared to manual tests, is the need for a human tester to continuously read the manual test steps in order to know what to input and what output to expect from the system. This factor also explains why test
experts are quicker, 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. Thus, in contrast to a human tester, the scripts only needs to execute their commands, i.e. removing the reading overhead completely. Additional gains are provided by the scripts ability to quickly paste textual input into the tested system, whilst a human has to write the input. These gains might seem small, but in the overall perspective, i.e. during execution of the entire test suite, these small gains add up and become significant.

Based on the collected metrics, the return on investment (ROI) for the automation can also be calculated by comparing the development time of the VGT suites and the manual execution time. Once again, these calculated results should only be compared between Case 1 and Case 2 if the contexts of these cases are taken into account. As shown in Table 5.7, the ROI in Case 1 would become positive after 14 executions of the VGT suite, i.e. the cost of executing the manual test suite 14 times equals the implementation cost of the VGT suite. Hence, after the 14 executions, the VGT suite would start providing a positive ROI. For Case 2, a positive ROI would be reached after 26 executions. However, since there is no cost related to running the VGT suites, they could be run at night, thereby greatly improving the test frequency and thereby provide almost daily feedback to the developers [21]. Hence, both VGT suites would reach a positive ROI within one calendar month, if executed every night, including weekends. This ROI calculation is however simplified, since it does not take the number of found defects into consideration, nor the cost for maintenance of the VGT suite. The maintenance costs of a VGT suite, developed in Sikuli, are however unknown, and therefore an important subject of future work. However, the industrial practitioners in Case 2 stated that they considered the maintenance costs, for the maintenance they were required to perform, to be feasible.

A graph visualizing when positive ROI is reached, conceptually, based on VGT transition cost and cost per manual test suite execution, is shown in Figure 5.8, supported by the model defined by Berner et al., 2005 [21]. This model assumes that the cost of the manual test case execution is linear, and that the maintenance costs are small and almost linear. Hence, the slightly sloped horizontal line starting at time $t_1$. Thus, positive ROI would be reached when the two lines cross, i.e. at time $t_N$ in Figure 5.8. It should also be noted that during the VGT transition period, $t_0$ to $t_1$, the VGT suite is not executed, whilst the manual test suite could be.
5.5 Discussion

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

5.5.1 Challenges, Problems, Limitations and Solutions

In this report we have presented challenges, problems and limitations (CPLs) identified in two industrial VGT transitioning projects. 58 CPLs were identified in Case 1, corroborated by findings from Case 2, related to different aspects of the transition, from CPLs in the tool to CPLs the tested system itself. However, despite the CPLs, the industrial practitioners from Case 2 report that they found VGT to be both a valuable and cost effective technique. In addition, they reported that even though the CPLs added a lot of frustration to the VGT transition they quickly found ways of mitigating the CPLs to avoid them during the transition process. Furthermore, the practitioners reported that they had not found any function that they could not automate using VGT. Similarly, the driving researcher in Case 1 found CPLs that were more problematic than others, but that most of them could be solved or mitigated. However, there were still a set of CPLs that required such huge effort to solve, i.e. changes to the development company’s test documentation process, that they became out of scope for the project. The CPLs that could not be solved related to the tested system itself, such as, lack of proper manual test case descriptions, the tested system either lacked functionality or contained a defect that caused the script(s) to fail every execution, etc. What the impact of these CPLs will be for future development of the VGT suite developed in Case 1 is still unknown and therefore a subject of future work. However, since the manual test cases also suffer from some of these CPLs, they are assumed to be solved as the tested system evolves, i.e. when the defects are corrected, and as the test documentation evolves.

Furthermore, there were several CPLs related to the VGT tool, Sikuli, but in contrast with the test system related CPLs, the Sikuli CPLs were reported to be solvable, or could be mitigated, using different practices, e.g. adding redundancies in the scripts. However, there are still CPLs connected to Sikuli that require addressing in the future, e.g. the volatility of the tool’s image recognition algorithm and the tool itself. However, this work lies outside the scope of our own research, since our objective is to evaluate VGT as a technique, and its industrial applicability, rather than any specific VGT tool. In addition, as discussed, Sikuli, the version currently available at the time this report was written, is highly volatile and prone to failure. These failures, and the lack of stability, is what has been most criticized with the tool when shown, or applied by, industrial practitioners. However, Sikuli is not the only available VGT tool on the market. In our previous study [37], we compared Sikuli with a commercial VGT tool, which showed to be much more stable, but was instead very costly. In addition, the fact that there are more stable tools out there, in combination with the positive feedback from industry regarding VGT, indicates that there is both a need and a want for the technique. However, as also reported in this paper, most CPLs collected during this project were related to the tested system, rather than the tool. Consequently, other
5.5. DISCUSSION

VGT tools will encounter these CPLs as well. Thus, more work is required on identifying general solutions that are applicable for all types of systems and VGT tools to mitigate or solve these CPLs.

Furthermore, this paper has discussed a small set of possible, generic, solutions to some of the CPLs, but these solutions are still discussed in the context of this study, i.e. VGT transition performed with Sikuli. Other VGT tools, e.g. JAutomate, do however have other properties than Sikuli. For instance JAutomate has another set of image recognition algorithms, and also support capture/record and replay of VGT scripts. Hence, the question becomes, which VGT tool is best? We do not think there is a definitive answer to this question, but rather that different VGT tools fit better or worse in different companies and contexts. However, a comprehensive comparative study of the available tools, preferably in an industrial context, is still a subject of future work. Furthermore, we do not conceive that VGT is the solution for all test related problems, it is only a compliment to previous manual and automated test techniques with the potential of relieving some of the tediousness and cost related to manual system- and acceptance-testing. Thus, VGT can not, and should not, replace a company’s current practices or tools, but rather compliment them [21]. Neither is VGT, even though it perceivably can emulate a human user, a replacement of human testers. Human testers will still be required, since the scripts can only identify faults within the scripted scenarios. However, since VGT relieves some of the testers’ workload, they can instead focus on more exploratory testing to uncover new and previously unknown faults. Thus, raising the quality of the tested system.

5.5.2 Defects and performance

In an ideal world, all industrial companies would have infinite amount of time to create perfect architectures, develop software according to well timed and planned processes and thoroughly test the software for defects. However, in the real-world these conditions seldom exist, with the obvious result that the software will contain defects. In the past, the primary way to uncover high system-level defects has been to use manual, tedious, and costly tests. Automated techniques do exist, e.g. record and replay, but studies have shown that they suffer from problems that have limited their industrial use. VGT has properties that mitigate these previous limitations, e.g. higher robustness to GUI layout change. However, in order for VGT to be industrially applicable it has to be able to uncover defects in the tested system. In this report we have presented information, from two projects in industry, which show that a manual test suite automated with VGT is not only able to identify all the defects the manual test suite can identify, but also new ones. In addition, the execution speed of a VGT suite allows the suite to be executed daily, whilst the industrial norm for manual system tests is once every three months to one year. Hence, a VGT suite can provide feedback to the developers about system-level faults with much higher frequency [21]. In addition, due to the close to non-existent cost of running the VGT suite, it can be rerun several times and execute the individual test cases in different order, thereby uncover new defects. However, the effects of randomized test case execution is still a subject of future work.
In our previous work, we automated approximately 10 percent of the test cases for another version of the tested system used in Case 1. Based on the collected data from our previous work we estimated that all the test cases would take roughly 160 hours to automate. However, based on the new data, the estimated time for automating all the test cases for the main product of the tested system would be 426 hours. Hence, 2.5 times longer than our previous estimates. However, the new development time is based on the development of a much more rigorous VGT suite, i.e. more robust, than in our previous work, e.g. including several levels of redundancy. In addition, our previous estimates were based on the development of only a few, yet well chosen, representative, test cases, but did not include any of the unexpectedly problematic ones identified in Case 1. Furthermore, the estimated speed-up between manual and automated test cases was also incorrect in our previous work. This faulty estimation can once again be related to the complex and time consuming scripts that were not covered in our initial study, e.g. test cases including large loops, and test cases requiring the user to just sit and wait for several minutes for a service to timeout. However, based on information collected in Case 2, we found a speed-up of a factor 16, compared with the marginal speed-up of 1.06 in Case 1. A conclusion can therefore be drawn that the performance increase of a VGT suite is also related to the manual test specification. Consequently, it is reasonable to say that in order to gain the highest performance possible from a VGT suite, one should be selective of which, and in what order, test cases are automated, e.g. not automating test cases that require longer waits.

5.5.3 Visual GUI Testing with Sikuli

Does Visual GUI Testing (VGT), performed with Sikuli, actually work? Based on the cheer number of challenges, problems and limitations (CPLs) that were identified in this study this question becomes highly valid. On the one hand we have the CPLs themselves, some of which, as reported by the leading researcher and the industrial practitioners, were both troublesome and added a lot of frustration to the VGT transition. On the other hand we have reports, from these same industrial practitioners and researcher, stating that VGT is both cost-effective and valuable for industrial use. In particular, the reports from the industrial practitioners are interesting, given that they performed the VGT transition in an industrial context with minimal support from the research team. Hence, providing information from a real-world context, with real-world constraints, e.g. time and cost constraints, of how the CPLs affected the transition and use of VGT in industry. However, the focus of Case 2, performed by the practitioners, was not on the CPLs, but rather on the transition itself. Furthermore, given that the transition project was driven by these practitioners, they may be biased in their evaluation. In addition, the information acquired from said practitioners was mainly qualitative, and only acquired as available since the data collection performed by the industrial practitioners lacked the rigor that is usually expected by academic researchers. However, Case 1, was driven academically, and supplied rigorous, detailed, information that corroborated the information supplied by the industrial practitioners. Hence, even though the VGT transition encountered many CPLs, a conclusion must still be drawn that the transition, and industrial use, of
5.5. DISCUSSION

VGT is feasible, flexible and cost-effective. However, further studies are still required to evaluate the maintenance costs, and long-term applicability of the technique.

5.5.4 Threats to validity

There are several threats to the validity of this study and its results. In terms of internal validity, this study was performed in two different companies, where the research team had full control over the VGT transition in one case, and almost no control in the other. The lack of control in the second case limited the academic rigor of the acquired information. In addition, the acquired information from Case 2 can still be biased, even though precautions were taken, i.e. separate interviews with the leading industrial practitioners and triangulation through analysis of documentation and inspection of the developed VGT suite. However, given that the majority of the information presented in this report was acquired in the case driven by a researcher, and only corroborated by the information supplied by the industrial practitioners, this threat is considered low. The experience of the leading research member in Case 1 is however also considered a threat, since the researcher had limited knowledge of VGT and of performing case studies. However, all data acquisition was supported by the more experienced research members and the analysis of the acquired information, which was substantial in size, and also rigor, was performed by another member of the research team. Thus, mitigating any positive or negative bias of the research member who performed the VGT transition.

Furthermore, the leading research member in Case 1 did not have any access to the information acquired in Case 2. Thus, mitigating the risk of said researcher being influenced by the reports from the industrial practitioners. In addition, both projects were performed on similar systems, in similar contexts, i.e. both safety-critical, with similar functionality. However, given that the systems were mutually exclusive, had manual test cases with different architecture, and the information collection practices differed, there is a threat that the collected information is completely incomparable between the two cases. For the general CPLs, this risk is considered small, but for the collected quantitative data this threat has been considered high. However, this has been addressed in the report by properly stating that said quantitative data should not be compared if the contexts of the two cases is not taken into account.

The external validity of the results, is in turn raised by the fact that two different industrial cases are used. Furthermore, given the different contexts, with a researcher in one case, and industrial practitioners in the other, which both provided similar, or identical information, the external validity of the results can be considered high. However, the information is still context dependent. Thus, may not be valid in smaller companies, for smaller systems, which are not safety- or mission-critical. Furthermore, since the leading researcher in Case 1 and the industrial practitioners in Case 2, where all inexperienced with VGT at the start of the project, the two cases provide comparable information in terms of experiences in learning and applying the technique. Hence, raising the external validity of the results for VGT transition projects in industry where little, or nothing, is known about VGT at the start of said project.

Yet another threat to the external validity is that both projects were per-
formed with the same VGT tool, i.e. Sikuli. Thus, limiting several of the conclusions to that tool, i.e. many of the CPLs and/or solutions do not apply to other VGT tools, e.g. JAutomate. Hence, lowering the external validity of the results.

Furthermore, the reliability of the study is considered high, given that it would be replicated in a similar context. The methodology, as well as the developed VGT scripts, and their architecture, have been described in full. In addition, given that the CPLs were identified within two mutually exclusive cases, i.e. with high external validity, the results leading to the identification of these CPLs should be replicable. We therefore welcome studies that replicate, and/or expand our results in other companies, contexts and/or with other VGT tools.

5.6 Conclusions

In this paper we present the results from two industrial projects, one driven by an academic researcher, the other driven by industrial practitioners, where manual system test suites were transitioned into automated Visual GUI Testing (VGT) suites, developed with the open source tool Sikuli. During the projects, information on challenges, problems and limitations (CPLs) of the VGT transition were collected. 58 CPLs, from which 28 mutually exclusive groups of CPLs, were identified as well as generic solutions to mitigate some of these CPLs. Furthermore, results on defect finding ability of the VGT suites, transition cost and return on investment (ROI) were identified. Analysis of the results show that VGT, despite the CPLs, is a valuable, cost-effective, flexible technique for system- and acceptance-test automation in industry.

The contribution of this study is significant since the currently most common practices for system- and acceptance-testing in industry are manual, thereby costly and tedious. Automated techniques, e.g. unit testing and capture/record and replay, have been suggested as the solution to mitigate the need for manual testing. However, these techniques have are in terms of functionality, flexibility and robustness, leaving a gap for a high-level automation technique for system- and acceptance-testing. VGT is perceived not to suffer from these limitations. By combining scenario based scripts with image recognition the technique becomes impervious to GUI layout change and is able to emulate end user behavior. However, the industrial applicability, including CPLs, of this technique were, before this study, still unknown.

In this paper, we present the results from two industrial VGT transition projects, during which 58 CPLs were collected, which were grouped into 28 mutually exclusive groups. Thus, contributing the to body of knowledge regarding VGT, but also the general body of knowledge of software engineering since it has been identified that studies focusing on the CPLs of techniques, methods and tools are sparse [45].

Analysis of the 28 groups of CPLs showed that most of the CPLs were related to the tested system itself, i.e. 34 out of 58, whilst only 14 were related to the VGT tool. However, the most prominent CPLs, which were summarized in the report, mainly relate to the tool, e.g. the volatility of the tool’s image recognition algorithm and the tool itself. Thus, raising the question if VGT is
at all applicable for industrial use? This question was answered by information from the two studied, independent, cases which stated that even though VGT suffers from many CPLs, it is still a valuable, cost-effective, flexible, technique for system- and acceptance-testing. Additional strength was given to this conclusion by the fact that this information was supplied by industrial practitioners performing the VGT transition, without academic expert support, in a real-world context, under real world time- and cost-constraints. Information that could then be corroborated by results collected through controlled, rigorous, academic information acquisition practices in the second case.

Furthermore, the applicability of VGT was supported by information regarding the techniques defect finding ability, which showed that not only could the developed VGT suite find all the defects the equivalent manual test suites found, but also additional defects. Hence, due to the low cost of executing the scripts, the VGT suites could be executed several time in sequence, with different order of the individual test cases, which identified previously unknown defects. Furthermore, analysis of the transition cost showed that positive ROI of the VGT transition could be achieved within one calendar month within both cases. This quick ROI was provided by the fact that the VGT suites can be executed daily, thereby providing the system developers with quicker feedback on an unprecedented level of system abstraction, i.e. system-level.

Consequently, this study shows that there are many CPLs related to the transition and usage of VGT, performed with Sikuli, which should be addressed in future research. However, the results also show that the technique provides a valuable, cost-effective, flexible compliment to companies current test toolboxes, with unprecedented support for system- and acceptance-test automation. Thereby answering the study’s main research question.

Acknowledgment

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

Paper E

JAutomate: a Tool for System- and Acceptance-test Automation
E. Alégroth, M. Nass, H. H. Olsson
EMSE pp. 460-465
Abstract

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

Manual testing is currently the primary approach in industry to perform system- and acceptance-testing. These tests are performed through interaction with the system under test’s (SUT) graphical user interface (GUI), on a regular basis, to ensure SUT conformance to the requirements, i.e. regression testing [13]. However, manual tests are error prone, tedious and time consuming, known to take up to 40 percent of the entire development budget of a project [51]. To mitigate these problems, test automation has been proposed as the solution, i.e. unit testing [17], record and replay [22–24], etc. However, these automation techniques have different limitations that restrict their use for system- and acceptance test automation, or make the automated tests costly to maintain. Hence, leaving a gap for a simple, high-level, test automation technique for system- and acceptance-testing.

Visual GUI Testing (VGT), as referred to by Börjesson and Feldt [37], is a tool driven technique that is perceived to cover this gap. VGT combines image recognition with scripts, for automated, scenario-driven, testing through a SUT’s GUI, which allows the user to automate tests on a high level of system abstraction and emulate end-user behavior. The technique is emerging in industrial practice, and empirical evaluation of VGT tools on industrial software has shown that the technique is industrially applicable [37]. However, the tools evaluated in this research required each script to be written manually, raising cost and the number of defects in the scripts.

In this paper, we present JAutomate, a VGT tool that mitigates the need for writing scripts by combining image recognition with record and replay functionality for quick, robust and flexible script development and playback. The paper presents the tool, its general properties, detailed properties of one of the tool’s image recognition algorithms, and a comparison of these properties to two other VGT tools based on information from previous research [37]. In addition, we perform an in depth multi-aspect analysis of the tools impact on a company’s business, architecture, process and organization (BAPO) [52]. The BAPO aspects were chosen to provide breath and depth to the analysis and show how the introduction of the tool will not just affect the adopting company’s test process but also other aspects of the company. Furthermore, we discuss the potential benefits of using JAutomate compared to manual system- and acceptance-testing and present the results from a survey, performed with 52 industrial practitioners, that shows not only an industrial need for the tool, but also that there are test-related problems in industry that the practitioners perceive can be solved, or mitigated, with the tool.

The specific contributions of this paper are:

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


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

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

System- and acceptance-testing is generally performed as a manual practice in industry due to the complexity and high level of system abstraction of the tests. The tests aim to test system conformity to the system requirements, and are performed regularly on the system under test (SUT), i.e. for regression testing [13]. Furthermore, the tests are defined in test specifications, which look different in different companies but with the same basic architecture based on scenarios defined by test steps, i.e. steps that define user input to the SUT for which there is some predefined output. However, this manual test practice is costly, time consuming, tedious and also error prone, requiring resources such as reference systems, testers, etc. Automated testing has been proposed as the solution to these problems, e.g. unit testing [17,18], model based testing [53], etc.

Unit testing [17,18], is a common test automation technique in industry which is performed on the SUT’s component level, i.e. to test low-level functionality of the SUT. This technique is therefore limited and unsuitable, due to complexity and maintenance costs, to use for higher-level tests, i.e. system- and acceptance-tests. These limitations can be mitigated using another commonly used technique in industry, i.e. record and replay (R&R) [22–24]. R&R is a tool-driven, two step, technique where user interaction with the SUT is first recorded in a script that can then be replayed automatically to perform regression testing. The recording can be done in several ways, e.g. using references to the SUT’s backend or by using exact coordinates on the SUT’s GUI. However, both of these methods suffer from limitations that, once again, require high maintenance, e.g. reference based R&R is fragile to API or even code change [26], whilst coordinate based R&R is fragile to GUI layout change. Hence, R&R does not fullfill all of industry’s needs for a robust, flexible, high system-level, automated test technique.

In the early 90s, Potter presented his tool Triggers [28], for system automation using image recognition. Other early work on automation using image recognition was performed by Zettlemoyer and Amant [29]. However, in recent years, the use of image recognition has also been transferred to testing, in tools such as JAutomate, Sikuli [27], etc. These tools use image recognition and scenario based scripts to perform tests through the SUT’s GUI, a technique that Börjesson and Feldt refer to as Visual GUI Testing (VGT) [37]. In their work, they provide support for VGT’s industrial applicability by comparing two VGT tools based on their static properties and the tool’s ability to automate manual, industrial grade, test cases for a safety-critical air traffic management system. VGT is perceived to resolve many of the limitations of previous techniques, i.e. R&R, because the technique is black-box [20], i.e. does not require any knowledge about the SUT. Additionally, it is robust to GUI layout change due to the image recognition. However, most VGT tools
require the test scripts to be written manually, which is associated with an up-front investment for the VGT transition, i.e. the test case automation. Consequently, VGT is perceived to be more flexible and robust than previous automation techniques, but the script development requires an upfront cost.

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

However, the introduction of JAutomate, like any new technique, method or tool, in a company, will affect several aspects of said company. There are several frameworks that capture these aspects, e.g. BAPO [52] and PESTEL [54]. BAPO, which stands for business, architecture, process and organization, was chosen as the framework for the analysis in this study because it provides a comprehensive high-level view of what will be affected during software development change in a company. Hence, the introduction of JAutomate in a company will potentially not just affect the testing process, but also the company’s business, e.g. raised quality can be used as a business advantage, architecture, e.g. architectural changes may be required to apply the tool, and organization, e.g. new roles may be required. Thus, BAPO was used to give both a broader, and deeper, analysis of the impact JAutomate can/will have at a company that chooses to adopt it for VGT.

6.3 JAutomate

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

JAutomate, as mentioned, is however not the only VGT tool available to the market, both open source and commercial alternatives exist. So what makes JAutomate preferable over the other available tools? In the following sections we aim to answer this question by first presenting a comparison of the static properties of JAutomate and two other VGT tools, i.e. Sikuli and CommercialTool (that for legal purposes will be kept anonymous in this report),
results shown in Table 6.1. The information for this comparison was acquired from our previous work [37] and complimented with information provided by JAutomate’s developer, Michel Nass. Secondly, we present a description of JAutomate’s perceived impact on the BAPO aspects of a company [52], based on previous academic empirical work with VGT, and Michel Nass’s expert experiences with JAutomate in industry.

6.3.1 Tool comparison

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

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

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

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

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

Image recognition sweeps per second. This metric is dependent on the computer the tool is executed on, but as reported by the official documentation
<table>
<thead>
<tr>
<th>Property</th>
<th>JAutomate</th>
<th>CommercialTool</th>
<th>Sikuli</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed in</td>
<td>Java</td>
<td>C#</td>
<td>Jython</td>
</tr>
<tr>
<td>Image recognition algorithm</td>
<td>Vizion Engine</td>
<td>Several algo-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rithms</td>
<td></td>
</tr>
<tr>
<td>Script language syntax</td>
<td>Custom</td>
<td>Custom</td>
<td>Python</td>
</tr>
<tr>
<td>Image representation in tool IDE</td>
<td>Text Strings /</td>
<td>Text-Strings</td>
<td>Images</td>
</tr>
<tr>
<td></td>
<td>images</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Image recognition sweeps per second</td>
<td>Depends on CPU(s) and image size</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Image recognition failure mitigation</td>
<td>Can automatically select from two different algorithms (color and contrast) or perform semi-automated test steps</td>
<td>Multiple algorithms to choose from</td>
<td>Image similarity configuration</td>
</tr>
<tr>
<td>Test suite support</td>
<td>Yes</td>
<td>Yes</td>
<td>Unit tests only</td>
</tr>
<tr>
<td>Remote SUT connection support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Remote SUT connection requirement</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cost</td>
<td>$849 per user per year</td>
<td>10.000 Euros per license per computer</td>
<td>Free</td>
</tr>
<tr>
<td>Record and Replay functionality</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Manual test step redundancy</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Semi-automated test steps</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Backwards</td>
<td>Yes (based</td>
<td>Guaranteed</td>
<td>Uncertain</td>
</tr>
</tbody>
</table>
of Sikuli and CommercialTool the tools can make approximately 5 and 7 sweeps per second respectively. No such metric is available for JAutomate since the image recognition algorithms are not only dependent on the performance, i.e. clock rate, of the CPU, but also the number of cores in the CPU, i.e. the algorithms’ performance scales linearly with the number of cores.

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

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

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

**Cost.** Sikuli is an open source product, meaning that it is free of charge. In contrast, CommercialTool and JAutomate Studio, i.e. JAutomate’s integrated development environment, are both commercial products that cost 10.000 euros per license per year and $849 per computer per year respectively. JAutomate’s runtime environment is however free of charge and can be used with a regular text editor together with a screen capture tool such as Snagit, etc.

**Record and Replay functionality.** Whilst both CommercialTool and Sikuli require the user to manually input the test scripts, JAutomate supports automatic recording of the test scripts. This functionality lowers the implementation time of the scripts, especially for scripts that are developed as 1-to-1 mappings to the manual test cases since these scripts can be recorded during normal execution of the manual test cases.

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

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

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

6.3.2 Business

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

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

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

6.3.3 Architecture

The architectural aspect can be viewed from two perspectives. First, the perspective of JAutomate, i.e. the tool itself, and second, from the perspective of the system under test (SUT). In the following sections, these two perspectives are discussed together with the benefits and drawbacks of different architectures.

6.3.3.1 The tool

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

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

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

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

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

6.3.3.2 The system under test (SUT)

Similar to all VGT tools, JAutomate considers the system under test (SUT) as a black-box [20], meaning that the tool does not require any knowledge...
about the SUT’s internal architecture, components, development language, etc. The black-box approach is achieved through the high-level interaction of the tool, i.e. through the image recognition that allows the tool to provide input and observe output through the SUT’s GUI. Consequently, all automated tests developed using JAutomate are executed in the same way as a human user would perform them, i.e. through the operating system’s internal methods which makes the tool’s interactions indistinguishable from a human user from the SUT’s perspective. In contrast, previous GUI-based test techniques, e.g. widget based record and replay, interact with the SUT by calling operations/methods within the SUT, i.e. white-box testing [20]. Hence, these interactions are performed below the GUI layer and thereby differentiate from end user interactions. In addition, some white-box tools require hooks, or interfaces, to be added in the SUT, thereby changing the internal architecture of the SUT. Hence, JAutomate, in contrast to white-box tools, is non-intrusive and does not require any changes to the SUT in order to be applicable. Furthermore, due to the GUI-based interaction, the tool can interact with any type of system that can be accessed manually from a computer client, i.e. it can test desktop, web and even mobile applications.

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

6.3.4 Process

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

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

6.3.5 Organisation

JAutomate is not considered to have a large impact on the organization, e.g. in terms of requiring new roles or redistribution of resources. The reason is because unlike other test techniques for complex testing, that sometimes require test automation specialists or experts to develop tests of high quality, e.g. in machine learning [55], JAutomate has been designed to be simple to use for any user. Thus, JAutomate has been developed to have high usability and learnability to allow both developers and testers with, or without, previous SUT or programming experience to use the tool. Hence, JAutomate does not require any change of a company’s current roles, introduction of new roles or infer replacement of human testers in their current roles. Instead, JAutomate is primarily a compliment to previous practices to make the testing more efficient in terms of cost and quality. In addition, it is perceived that VGT tools, e.g. JAutomate, help alleviate the tediousness of performing manual tests over and over. Tediumess that lead to developers and testers taking shortcuts, making mistakes, etc., which lowers the quality of the testing, and in extension, software quality. Consequently, the impact on a company’s organization is low but high on the company as a whole.

6.4 The industrial need

In order to investigate the industrial need for JAutomate, and VGT in general, a survey was performed during a seminar about the technique, attended by approximately 100 industrial practitioners. The survey had two purposes. First to identify the test related problems that Swedish software development companies are currently facing. Second to evaluate the industrial practitioners’
6.4. THE INDUSTRIAL NEED

knowledge about VGT, and JA Automate, and if VGT perceivably could solve, or mitigate, some of the test-related problems experienced by the practitioners. The first purpose was investigated through a question where the industrial practitioners got to distribute 100 points to rank 17 predefined test-related problems. 52 questionnaires were collected, hence a response rate of roughly 50 percent. Figure 6.4 visualizes the results from the first question, shown as the percentage of distributed points over the test-related problems.

Both the highest ranked and the third highest ranked problems, i.e. test-related cost and testing to late, can perceivably be mitigated by using VGT since VGT scripts can be executed without additional cost which makes it possible to raise the test frequency. Additionally, the problem with insufficient customer feedback can be mitigated by recording customer acceptance tests, mitigating the need for having customers on site for acceptance regression testing. Problems regarding test coverage can perceivably also be mitigated since VGT releases resources required for manual testing, i.e. developers and testers. These resources can instead be used to perform exploratory testing of the system to uncover previously unknown faults. Hence, VGT, performed with JA Automate, has the potential to solve or mitigate several of the largest test related problems encountered in industry.

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

Visual GUI Testing (VGT), with tools such as JAutomate, have been designed for development of high system-level tests for automated regression testing. Automated regression testing is generally proposed to be a good practice and has been incorporated in agile development processes such as eXtreme Programming to facilitate continuous integration (CI) [56]. However, CI testing is mostly associated with XUnit testing [17], but, as proposed by Fowler [56], other tools for end-to-end testing should be incorporated into the test process as well, e.g. FitNesse [57]. JAutomate has been designed with CI in mind and the test scripts can even be exported and run in FitNesse, or stand alone as a compliment to other automated testing. Hence, JAutomate is, and was designed, to be a compliment to other automated testing in a CI context, i.e. providing support for automated system- and acceptance-testing that previously had to be performed manually.

Furthermore, this report has presented JAutomate as a testing tool to perform VGT. However, given the capabilities of the tool, e.g. its ability to interact with any GUI based application, the tool can also be used for automation, e.g. to automate the build process performed during CI to minimize the risk of erroneous builds due to complexity [56]. In addition, JAutomate opens up new possibilities for monitoring of systems where it is unfeasible to use humans, e.g. to monitor memory usage during long-time tests or tests where input is given continuously to the SUT for longer periods of time, e.g. for 24 hours straight. Furthermore, JAutomate, and VGT, is perceived to be able to test non-functional properties of a SUT, e.g. usability, performance, etc, but no study has been performed to validate this claim, which is a subject of future research.

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

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

6.6 Conclusion

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

The industrial need to lower lead times and raise software quality are ever growing, pawing the way for new research into automated techniques and tools for all aspects of software engineering, from requirements engineering to testing. Test related automation techniques approach testing on different levels of system abstraction, from system- to code-level. However, due to different limitations of these techniques and tools there is currently a gap for a high-level, cost-effective, flexible and robust tool for system- and acceptance-test automation.

JAutomate fills this gap, with simple record and replay functionality, combined with image recognition, which allows the user to automate all types of user interaction performed through the system under test’s (SUT) GUI. A comparison between JAutomate and two other VGT tools shows that JAutomate has several benefits over the other tools, but also that the tools have different properties that make them suitable in different contexts. In addition, we have presented how JAutomate will have beneficial impact on other aspects than just the test process. For instance, from a business aspect, JAutomate will help raise software quality, which can be a business advantage. Additionally, since the tool is black-box it does not require any knowledge or modification to the SUT’s architecture in order to be applicable. Furthermore, the tool is designed with both novice and advanced users in mind and does therefore not affect the company’s organization, e.g. by requiring new, or changed, roles.

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

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


