Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study

Emil Alégroth
Software Eng. and Tech.
Chalmers University
Gothenburg, Sweden
emil.alegroth@chalmers.se

Zebao Gao
Dept. of Computer Science
University of Maryland,
College Park
MD, USA, 20740
gaozebao@cs.umd.edu

Rafael A.P. Oliveira
ICMC/USP
University of Sao Paulo
Sao Carlos, Brazil
rpaes@icmc.usp.br

Atif Memon
Dept. of Computer Science
University of Maryland,
College Park
MD, USA, 20740
atif@cs.umd.edu

***Abstract*—In this paper we present the results of a two-phase empirical study where we evaluate and compare the applicability of automated component-based Graphical User Interface (GUI) testing and Visual GUI Testing (VGT) in the tools GUITAR and a prototype tool we refer to as VGT GUITAR. First, GUI mutation operators are defined to create 18 faulty versions of an application on which both tools are then applied in an experiment. Results from 456 test case executions in each tool show, with statistical significance, that the component-based approach reports more false negatives than VGT for acceptance tests but that the VGT approach reports more false positives for system tests. Second, a case study is performed with larger open source applications, ranging from 8,803-55,006 lines of code. Results show that GUITAR is applicable in practice but has some challenges related to GUI component states. The results also show that VGT GUITAR is currently not applicable in practice and therefore requires further research and development.**

Based on the study's results we present areas of future work for both test approaches and conclude that the approaches have different benefits and drawbacks. The component-based approach is robust and executes tests faster than the VGT approach, with a factor of 3. However, the VGT approach can perform visual assertions and is perceived more flexible than the component-based approach. These conclusions let us hypothesize that a combination of the two approaches is the most suitable in practice and therefore warrants future research.

***Keywords*-Component-based testing; Visual GUI Testing, GUI Testing; Test automation, GUITAR.**

I. Introduction

Modern Graphical User Interface (GUI) applications are complex, run on multiple devices (with different output capabilities), are multi-language, and use a combination of native (e.g., buttons, menus) and advanced (custom) widgets. The modes of interaction with these applications are also evolving, from button clicks to gestures. These advances in technology combined with the need for flexibility create challenges for current GUI testing (or system testing via the GUI).

Current GUI based test approaches and harnesses (tools) have different advantages and disadvantages in practice. Component-based GUI testing, with tools such as GUITAR [1], is associated with robust, automated and fast test execution but is also limited to Applications Under Test (AUT) written in specific programming languages. Visual GUI Testing is applicable to any GUI driven AUT, due to the use of image recognition, but is instead associated with slow and fragile test execution [2]–[4].

Because of these advantages and disadvantages, we hypothesize that there are characteristics of applications (containing custom widgets) and testing needs (e.g., multi-platform, multi-language) that lend themselves to a combined component-based and VGT based solution. Characteristics that also connect to different types of defects on different levels of AUT abstraction, from the pictorial GUI, to the GUI model, to code logic. In this paper, we provide support for the above stated hypothesis from an experiment and a single, holistic, empirical case study [7].

In the experiment, we apply GUITAR and a prototype tool referred to as VGT GUITAR to 18 faulty versions of a GUI driven AUT, developed by means of mutation operators [8], to compare the test approaches' fault finding ability and false result frequency. Analysis of the experimental results show, with statistical significance, that the component-based approach reports more false negatives for acceptance tests but the VGT approach reports more false positives during system testing. Thereby supporting the hypothesis that a hybrid approach could be the most suitable dependent on contexts and test purpose. A hypothesis with only limited previous support [6].

In the sub-sequent case study, the two tools are applied on three larger, open source, AUTs to evaluate the tools' current applicability in practice. Results show that GUITAR is applicable but has challenges related to component states that cause one third of the tests to report false positive results. Further, due to immaturity, VGT GUITAR is not applicable in practice. Based on these results we outline areas of future work for both the tools and their approaches.

The specific contributions of this work are as such:

1) Comparative, empirical, results regarding the applicability and defect-finding ability of automated component-

978-1-4799-7125-1/15/$31.00 ©2015 IEEE

based and VGT based testing; and
2) Support for the hypothesis that a combination of the component-based approach and the VGT approach is suitable in practice.

The continuation of this paper will be structured as follows. First, in Section II we present a background and an extended motivation to the necessity and importance of this work. The research methodology is then described in Section III followed by the research results and analysis in Section IV. These results are then discussed in Section V, followed by related work in Section VI and finally we conclude the paper in Section VII.

II. Background and Motivation

Market demands on software developers to rapidly produce new software and add new features to existing software create new challenges for software quality assurance. Challenges that have been proposed as solvable with automated testing [10], [11]. However, software is becoming more GUI intensive with innovative means of interaction, functionality and components. Consequently challenging the capabilities of current GUI test harnesses and presents a need for new automated GUI based testing approaches.

Alégroth et al. [5] classify the existing approaches to GUI based testing in three chronological generations. The first generation uses exact coordinates that are recorded during manual interaction with the AUT and automatically replayed for regression testing [12], [13]. However, this approach is associated with high maintenance costs due to lack of robustness to GUI change, dependence on screen resolution, etc., and has therefore been abandoned.

Second generation GUI based testing tools operate directly against the GUI model through GUI Widgets, component properties and values, which makes the approach more robust to AUT change, has high test execution performance, stable test execution, supports test recording but also GUI ripping and automated test case generation with tools such as GUITAR [1]. However, to access the GUI model, second generation tools require access to the AUT's underlying GUI library. Thereby restricting the tools' applicability to AUTs written in specific programming languages with known GUI APIs, non-distributed systems, etc [5].

Third generation GUI based testing, also referred to as Visual GUI Testing (VGT), uses image recognition in order to interact and assert AUT correctness through the pictorial GUI as rendered on the user's monitor. However, even though the industrial applicability of the approach has been shown [5], there are still gaps in knowledge regarding the approach long-term viability in practice and the approach is associated with robustness problems [4]. Furthermore, only initial research has been presented regarding test generation of VGT test case [9].

The stated benefits and drawbacks about the second and third generation approaches let us hypothesize that there are potential synergy effects between them, i.e. that by combining them the strengths of both techniques could be used. To the authors' best knowledge, Unified Functional Testing (UFT) is the only currently available tool that supports both component-based and VGT-based test scripts [14]. However, UFT does not support test case generation and is therefore associated with costs for manual script development and script maintenance. Thus, presenting an industrial need for this work, also supported by related work [15].

This paper presents a comparative study where we evaluate contextual differences that make the component-based approach beneficial to the VGT approach and vice versa. The study results were acquired with GUITAR and a prototype tool, referred to as VGT GUITAR that combines the model-based automated testing from GUITAR with image recognition from a script engine written in Python and the Sikuli API [16]. VGT GUITAR's architecture is shown in Fig. 1. The prototype uses an extension of GUITAR's ripper to acquire screenshots (bitmaps) of the AUT's components that are taken complementary to the ripping of the GUI component properties. In addition, the tool includes a purely bitmap based ripper that extracts component bitmaps from screenshots of the entire AUT's GUI taken during replay of the GUITAR test cases. The prototype also relies on GUITAR's test case generator that in turn has been extended with a filter function that removes all test cases that cannot be executed by VGT GUITAR. Filtering is performed by comparing the pool of acquired bitmaps from the AUT's GUI against the bitmaps explicitly required to drive each test case. If a test case includes a test step for which no bitmap was identified during ripping, the test case is removed. Filtering is required since the current tool prototype only captures bitmaps for basic Java components, e.g. JButton, JTextField, etc. GUITAR on the other hand can interact with a larger set of components. The filtered test cases are then executed through the VGT script engine that produces test results that for each test step present if the interaction and assertion of the GUI event passed or failed. As such, VGT GUITAR performs an implicit assertion if the component exists on the screen before it interacts with it, followed by an assertion of the AUT's behavior in the output assertion.

III. Methodology

The methodology used in this work was divided into an experiment and a single, holistic, case study with two units of analysis [7], i.e. component-based and VGT testing. The experiment was designed to evaluate the two approaches ability to find different types of defects. These defects were seeded into a small, yet representative, application created by the research team. In the case study, the two tools were applied on three open source applications to evaluate the tools' current applicability in practice. The following subsections will present the research design in more detail.

A. Experiment: Fault detection and False results

In order to compare the defect finding ability of the two GUI based test approaches we used the concepts of mutation testing to create 18 GUI level mutation operators. Mutation operators [17] are used for mutation testing to create slightly different versions through manipulation of the AUT's code.

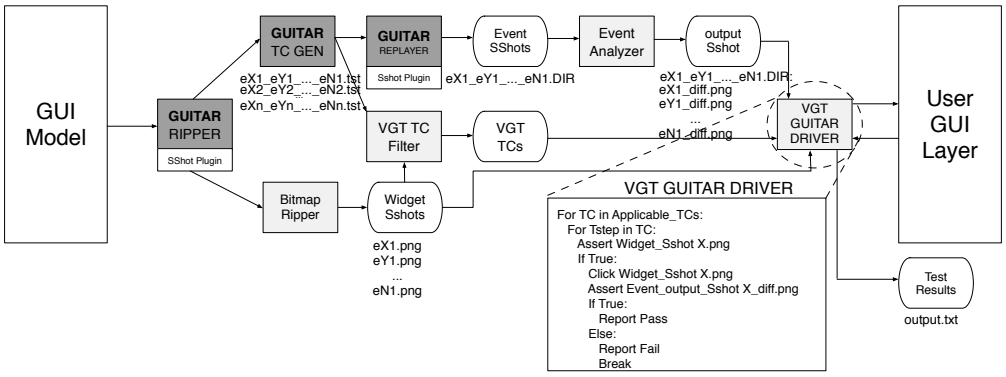


Fig. 1: Visualization of VGT GUITAR's architecture. Squares show code components of the tool and rounded squares show artifacts generated by said components.

For each mutation operator, shown in Table I, hypotheses were formulated regarding the expected behavior of GUITAR and VGT GUITAR when applied for regression testing on a mutant created with said operator. Hence, these hypotheses describe the test outcomes that were expected prior to the study based on the tools' individual capabilities. The hypotheses have been presented in columns four and five of Table I[1]. As an example, Mutant Operator 1 specifies the removal of a component from the GUI, which is a common, intentional or unintentional, occurrence when software evolves. The hypothesized correct behavior of both tools in this instance is that any test case that requires interaction with the removed component will fail.

Further, in order to identify if a reported failure was correct, a false positive or a false negative result, we also hypothesized how the mutation operator would impact system and acceptance testing of the AUT. System testing was defined as an assertion of the correctness of the AUT's functionality/behavior, i.e. input and output, and acceptance testing was defined as an assertion of the correctness of the AUT's functionality/behavior and its appearance. Note that we in this context assume that the mutation occurred unintentionally rather than as a planned change to the AUT, i.e. the mutations are regarded as regression defects. As an example, for Mutant Operator 1, if a GUI component is not rendered during testing we expect a human to fail both a system test regarding the AUT's behavior as well as an acceptance test, which also takes the AUT's usability, appearance, etc., into account. The system and acceptance testing hypotheses are stated in the two last columns of Table I.

To test the hypotheses, the research team created a custom Java application, referred to as AppX, with a simple, modifiable, design but with intricate enough functionality to host all the mutants. AppX has an MVC (Model-View-Controller) architecture with two input buttons and two output fields that display the output, as shown in Fig. 2. The top output field is a JTextField that can be asserted through the textfield's property values but the lower output field is a custom panel that renders the output with green letters on a black background that can only be asserted visually. This GUI design was chosen to make the comparison of the two approaches' capabilities fair by ensuring that both techniques could assert the correctness of the AUT. Hence, GUITAR can perform assertions through the top output field's GUI component values and VGT can visually asserts the rendered output in both the top and lower output fields. The rationale behind using AppX rather than an open source AUT for the experiment is to limit confounding factors and to have control of the mutant creation. We also claim that the external validity of the acquired results are high, despite the AUT's smaller size, since the compared test approaches operate on the highest levels of system abstraction which effectively transforms the AUT's business logic into a black box that we stimulate with GUI input to acquire specific GUI output. As such, the size and complexity of the AUT's business logic is irrelevant for the comparison of the two approaches ability to find GUI level regression defects.

To provide further support for our claim about the external validity the reader is referred to Fig. 3. In the figure, box size represent the depth of business logic in an AUT. At the top of the figure, the business logic of an application with a deterministic implementation of the traveling salesman problem is shown. The theoretical application takes a list of graph nodes, Xs, as input through the GUI and then displays an ordered list, Ys, with the nodes sorted the way they should be optimally traversed. Since the algorithm is deterministic it will always return the same ordered list, Ys, for a given input Xs. In the bottom of the figure we visualize the logic of an AUT where the traveling salesman algorithm has been mocked to a single static method that takes the same list, Xs, as input and returns the correct ordered list Ys as output without any computation. From a user's perspective the behavior of the two applications, on a GUI level, are equivalent, which infers that simplified applications can be used to simulate how GUI level tests behave also on applications with advanced business logic. If the application in the bottom of Fig. 3 is changed/mutated such that it displays another list Zs, where Zs $\neq$ Ys, it would appear to the user as a defect in the business logic in the application at the top of the figure. Hence, as a defect in the

[1]All resulting mutants from the experiment can be found online. See: http://www.labes.icmc.usp.br/∼rpaes/ICST2015Data/ICSTMutatedGUIs.html

TABLE I: Mutation operators and hypotheses of the results of applying each of the operators on AppX. **op.** - operator, **hyp.** - hypothesis, **Sys.** - System, **Acc.** - Acceptance.

#	Type	Mutation Operator	VGT hyp.	Comp. hyp.	Sys. test	Acc. test
1	Rem.	Remove completely	F	F	F	F
2	Rem.	Invisible	F	F	F	F
3	Rem.	Remove listener	F	P	F	F
4	Dup./ Ins.	Add identical widget	F	P	P	F
5	Dup./ Ins.	Add similar widget	F	P	P	P
6	Dup./ Ins.	Add different widget	P	P	P	P
7	Dup./ Ins.	Add another listener	P	P	F	F
8	Mod.	Expand/Reduce size of windows and widgets will auto-adjust their sizes	F	P	P	P
9	Mod.	Expand size of windows and widgets will not auto-adjust their sizes	P	P	P	P
10	Mod.	Reduce size of windows to hide widgets	F	P	F	F
11	Mod.	Modify location of a widget to a proper location	P	P	P	P
12	Mod.	Modify location of a widget to edges of windows	F	P	F	F
13	Mod.	Modify location of a widget to overlap with another	F	P	F	F
14	Mod.	Modify size of widgets	F	P	P	P
15	Mod.	Modify appearance of widgets	F	P	F	F
16	Mod.	Modify type of widgets (Button changed to TextField)	F	F	F	F
17	Mod.	Modify GUI library for widgets (Swing button changed to AWT Button)	P	F	P	P
18	Mod.	Expand/Reduce size of windows and widgets will auto-adjust their sizes	F	P	P	P

logic that could have been introduced, for instance, during maintenance of the application, i.e. a regression defect. As such, the results acquired during the experiment with AppX are perceived to have the same external validity as results acquired for GUI based regression testing of a larger application with similar types of GUI components.

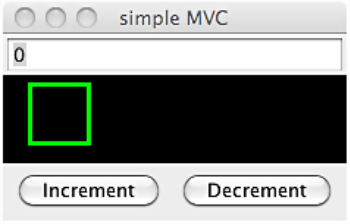


Fig. 2: Original GUI for AppX.

Another rationale for the use of AppX was to remove confounding factors from the experimental results caused by defects or other uncontrollable implementation related aspects of another application. By keeping AppX small we could be more certain that the results we acquired were caused by, and only by, our manipulation of AppX. An alternative had been to perform the experiment on several different applications and through randomization remove the impact of implementation specific problems, which is therefore a potential subject of future work to verify our results.

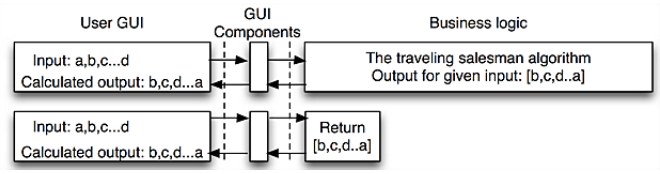
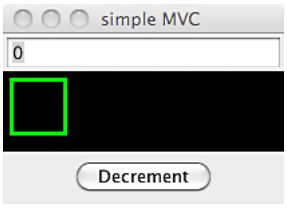
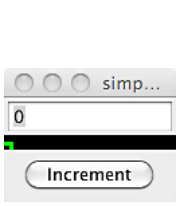


Fig. 3: Visualization of two applications with different business logic complexity but with the same behavior for a given input.

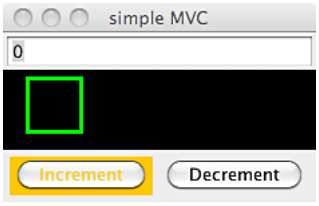
The mutation operators in Table I were manually seeded to create 18 different versions of AppX. For instance, for mutation operator 1 the increment button was removed from the view class and the listener from the control class. Fig. 4 shows three mutated instances of AppX, to be compared to the original GUI in Fig. 2. The seeding process was performed systematically for each operator by changing as few lines of code as possible for the instance to manifest the mutant. Each instance of AppX was then tested manually to ensure that the introduced mutant had not affected other aspects of the AUT's functionality.



(a) 1^{st} Mut. Operator.



(b) 10^{th} Mut. Operator.



(c) 15^{th} Mut. Operator.

Fig. 4: Three mutated instances of AppX.

Once the 18 instances had been created, a complete/holistic test suite was created for the original version of AppX that was executed three times on each instance to evaluate the average number of identified defects, false positives, false negatives and test case execution time. The test result was classified as correct if the test case terminated after completing the test case or if it terminated correctly after it had identified a mutant. The test result was classified as a false positive if the test case failed when it was expected to pass and as a false negative if it passed but it was expected to fail. Categorization of the test results was performed through manual inspection of the

test results in comparison to the expected behavior as stated in Table T1. As such, eight sets of results were acquired for:

1) False positive component-based system tests;
2) False positive VGT-based system tests;
3) False negative component-based system tests;
4) False negative VGT-based system tests;
5) False positive component-based acceptance tests;
6) False positive VGT-based acceptance tests;
7) False negative component-based acceptance tests; and
8) False negative VGT-based acceptance tests.

The stated hypotheses were considered supported if all applicable test cases of the test suite conformed to the expected, predefined, test behavior, e.g. if the hypothesis was that GUITAR would report failed test cases for a certain mutation, all test cases affected by the mutation had to fail in order for the hypothesis to be considered supported. Note that in cases where the mutation was only applied to one GUI component of the AUT, test cases that did not interact with said component were assumed to pass as in the original version of AppX.

The results of the experiment were then analyzed using formal statistics with the non-parametric Wilcoxon rank-sum test to identify, with statistical significance, if there was any difference between the two test approaches' test results. The non-parametric test was used because normality analysis with the Andersson-Darling normality test showed that none of the test result data sets were normally distributed.

B. Case study: Applicability in practice

The second part of the evaluation of the two approaches was performed as a single, holistic, case study with the purpose to evaluate the component-based and VGT-based approaches' applicability in practice. Note that use in practice, in this context, does not imply industrial context but for open-source software that infers, but does not show, usability in industrial practice. Due to VGT GUITAR's dependence on the GUITAR ripper and replayer, the case study was performed with three Java applications, which properties have been presented in Table II. These applications were chosen from an existing pool of applications that have been used in previous research to evaluate GUITAR's applicability [18]. Whilst the size of the applications in lines of code is considered small, compared to applications in industrial practice, the applications have rich GUI's with many components and events that, according to the reasoning presented in Section III-A, provide results with a high degree of external validity to industrial grade applications.

The case study was categorized as a single holistic case study with two units of analysis, being the component-based and VGT approaches represented by GUITAR and VGT GUITAR [7]. We do not classify the study as an experiment because, even though we compare the tool's results descriptively, the purpose was to explore the individual applicability of the tools.

The study was performed by constructing and executing test suites of 1000 randomly selected test cases for each AUT in GUITAR. These test suites were then filtered, as explained in Section II, to create test suites executable with VGT GUITAR.

TABLE II: Summary of the properties of the chosen open source applications that were used during the case study.

Application	Version	LOC	# of Windows	# of Events
JEdit	5.1.0	55.006	20	457
Buddi	3.4.0.8	9.588	11	185
Rachota	2.3	8.803	10	149

In order to improve the internal validity of the results, the results were captured as the average results over two runs of the two tools for each of the three applications.

The key metrics that were measured during the case study were:

1) The average percentage of VGT executable test cases out of the 1000 generated tests;
2) The average execution time per test case for the two approaches; and
3) The percentage of failed test cases per tool and per application.

A qualitative analysis was then performed of the collected metrics to identify the tools perceived applicability in practice. Additionally, the metrics were compared to test the hypothesis that the applicability of GUITAR is higher than the applicability of VGT GUITAR. Further, the hypothesis that the VGT GUITAR prototype is not currently ready for use in practice was also tested. The rejection criteria of the latter hypothesis were that the number of correctly terminating test cases would be significant in number and that the test execution time would be reasonable.

IV. Results and Analysis

The following subsections will present the results acquired from the experiment and the case study presented in Section III.

A. Experiment

The main results from the experiment are summarized in Table III. The table shows the number of false positives and false negatives reported by the component-based approach (GUITAR) and the VGT approach (VGT GUITAR). False results are reported as the average number of false results over three runs of each test suite, i.e. for each version of AppX. Three runs were performed to eliminate any non-determinism caused by Sikuli's image recognition algorithm. As such, a zero in the table indicates that all the test cases in that run terminated correctly, i.e. either found the mutant or did not report a mutant when there was none to report. These results were then analyzed using formal statistics where an Adersson-Darling normality test was first applied that showed that none of the samples were normally distributed. Because of the lack of normal distribution a non-parametric two-sided Wilcoxon rank-sum test was chosen to compare the samples at a 95 percent confidence level, results shown in Table IV.

The results of the statistical analysis show that there was no statistical significant difference between the two techniques in terms of reported false positives for acceptance tests or

false negatives for system tests. However, in regard to false positives, the VGT approach will report statistically significantly more false results for system tests, meaning that the approach will fail test cases that should have passed. The cause of this result is that the VGT approach requires the GUI components of a new version of the AUT to have similar appearance as the original AUT and requires the components to be visible on the screen. In contrast, the component-based approach, because it interacts with the GUI model rather than the pictorial GUI, can access components that are hidden or have changed appearance. For instance, for the 10^{th} mutation operator, shown in Fig. 4b, the AUT's window was reduced, obscuring the decrement button. GUITAR could identify the button through the GUI model and interact with it, testing the systems functionality. However, VGT failed the same test case because the image recognition could not find the button, in the same way as a human would during an acceptance test. As such, if the purpose of the test is to exercise the functionality of the system regardless of its graphical appearance the component-based approach is more suitable.

However, the acquired results also show that the component-based approach reports statistically significantly more false negatives for acceptance tests. Hence, the approach fails to report defective AUT states that a human would report, such as defective AUT behavior, un-interactable GUI, etc. The cause of this result is the same as for the previous stated result, i.e. that the component-based approach can interact with hidden or incorrectly rendered components because the GUI's pictorial appearance is omitted from the assertions. Consequently, the VGT approach is more suitable for automated acceptance testing where both the AUT's functionality and appearance need to be considered to satisfy customer needs.

Special attention should be given to mutated versions 3, 4 and 17 of AppX, mutant operators presented in Table III. In version 3, one of the GUI component listeners was removed, leaving the GUI component visible, and interactive, but it did not generate any output. The component-based approach, in this case, was not able to identify that no new output was produced and therefore reported false negative results both for a system and an acceptance test. In contrast, due to VGT GUITAR's ability to assert both input and expected output visually, it could detect the faulty AUT behavior.

In version 4, an extra increment button was added to the AUT that caused the VGT approach to reported false positive results both for system and acceptance tests. The reason was because of the image recognition algorithm's behavior, which sweeps the screen from the top left to the bottom right and starts from the position where the last match was found. Thus, the tool clicked on the correct increment button half of the time, thereby providing support to previous work regarding the lack of robustness of the VGT approach [5]. The component-based approach did not have this problem because the properties of the original increment button remained the same, allowing it to be identified.

The third case of interest was for version 17, where the AUT's GUI library was changed from Swing to AWT (Abstract Window Toolkit). Consequently changing the properties of the GUI components but not their appearance. This change caused problems for the component-based approach that reported false positives for all test cases for both system and acceptance tests. In contrast, the VGT approach successfully executed all test cases. Thus providing support for the VGT approach flexibility compared to the component-based approach.

Out of the 18 stated hypotheses, shown in Table I, three results deviated from the expected result. These deviations occurred for mutation operators 4, 5 and 7. In 4, we hypothesized that VGT would fail in all instances when an additional, equivalent, button was added to the GUI. However, as discussed, only some test cases failed due to the image recognition algorithm's sweep pattern. In 5 we also hypothesized that the VGT approach would fail, i.e. that adding a similar button would cause problems for the image recognition. However, in this case all test cases behaved accurately. Thus proving the hypothesis wrong. In 7 we hypothesized that VGT would succeed even though an additional listener was added to one of the AUT's buttons. However, because of the listeners impact on the output result, VGT GUITAR's output assertions failed, which from a test point of view was correct behavior but rejected our hypothesis. For all other mutation operators our hypothesis of expected behavior were supported by the results, for both test approaches.

456 test cases were executed during the experiment, in each tool (912 in total), with a total execution time for GUITAR of 591 seconds (on average 4.1, standard deviation 0.2 seconds per test case) and 3321 seconds for VGT GUITAR (on average 23, standard deviation 10.7 seconds per test case). As such the execution time for the VGT approach is significantly higher, which can be explained by the speed of the image recognition and because assertions were performed for both input and output to/from the AUT.

B. Case study

The collected metrics from the second part of this study, i.e. the case study, have been summarized in Table V.

In order to get suitable samples, test cases were generated in the order of 100k per application from which the test suites of 1000 test cases were randomly selected. The order of magnitude was controlled by the test case length, which due to state space explosions were kept between two to four steps per test case depending on application. Trial runs helped evaluate a suitable number of test case steps for each application and it was found that, as an example, over three million test cases was produced for the JabRef application of length three. Test generation of all three million test cases was deemed unfeasible since the execution time would have been many hours. Instead we opted for a shorter test case length, e.g. length two for JabRef. However, this decision also impacts the test cases representativeness for test cases used in industrial practice that are generally based on intricate user scenarios designed to test specific features of the AUT. As such, we stress that the results of this study are indicative of the tools' current applicability in practice but that more work is

TABLE III: False results in the context of system and acceptance tests from the experiment with AppX. **G.** - GUITAR, **V.** - VGT GUITAR, **FP** - False positive, **FN** - False negative, **ST** - System test, **AT** - Acceptance test, **FAIL.** - Failure.

#	G. FP ST	V. FP ST	G. FP AT	V. FP AT	G. FN ST	V. FN ST	G. FN AT	V. FN AT	G. FAIL. CAUSE	V. FAIL. CAUSE
1	0	0	0	0	0	0	0	0	-	-
2	0	0	0	0	0	0	0	0	-	-
3	0	0	0	0	0	0	7	0	Rendered output	-
4	0	3	0	3	0	0	0	0	-	Several identical widgets
5	0	0	0	0	0	0	0	0	-	-
6	0	0	0	0	0	0	0	0	-	-
7	0	0	0	0	0	0	8	0	Rendered output	-
8	0	8	0	8	0	0	0	0	-	Widget size
9	0	0	0	0	0	0	0	0	-	-
10	0	8	0	0	0	0	8	0	Ignore GUI appearance	Hidden widgets
11	0	0	0	0	0	0	0	0	-	-
12	0	7	0	0	0	0	7	0	Ignore GUI appearance	Hidden widgets
13	0	7	0	0	0	0	7	0	Ignore GUI appearance	Hidden widgets
14	0	7	0	7	0	0	0	0	-	Widget size
15	0	7	0	7	0	0	0	0	-	Widget appearance
16	0	0	0	0	0	0	0	0	-	-
17	8	0	8	0	0	0	0	0	GUI library change	-
18	0	8	0	0	0	0	8	0	Ignore GUI appearance	Widget size change

TABLE IV: Results of the statistical analysis of the results presented in Table III with a non-parametric two-sided Wilcoxon rank-sum test performed at a 95 % confidence interval.

	FP ST	FP AT	FN ST	FN AT
P-Value	0.01236	0.1883	n/a	0.009044
Result	Reject H0	Accept H0	Accept H0	Reject H0
Meaning	VGT reports more false positive system test results	No difference between test approaches for false positive acceptance tests	No difference between test approaches for false negative system tests	Component-based testing reports more false negative acceptance test results

required to show their usability in industrial practice. Further, we also stress that the state space explosion for GUI level test case generation is a problem that requires future research and development in order to raise the applicability of the tools in industrial practice.

The GUITAR test suites of 1000 test cases for each of the applications were then filtered to extract the test cases that could be executed with the VGT approach, i.e. the VGT GUITAR prototype. Table V shows that the number of VGT applicable test cases was on average less than five percent of the 1000 generated test cases. This low number of test cases is directly influenced by the prototype's current ripper function, explained in Section II, which is unable to adequately capture images of all of the GUI's components. As such, further development of the prototype is required to improve upon the bitmap ripping.

After execution of the VGT based test cases, it was found that all of them had failed during execution. A failure was in this context identified as a false positive caused by erroneous test case scenarios, identified through visual inspection and manual replication of the test cases. Erroneous test scenarios are in this context scenarios that interact with components that are not visible on the screen but that can be accessed through the GUI model, e.g. menu items that GUITAR can access without opening the menu. Other failures were caused by GUI components being in disabled or faulty states, e.g. disabled buttons. The root cause of these problems is that the test generation is performed without taking user interaction with the AUT into consideration, i.e. the test cases lack domain knowledge since they are created from stateless Event-flow graphs (EFG) that do not ensure event availability at runtime. Omission of state information in the EFGs is a deliberate design decision taken to improve scalability of the EFGs for larger and complex applications with many events and states. To verify that the lack of context information in the stateless model was the source of the failing VGT GUITAR test cases, test cases were created manually for all three applications, using images ripped by GUITAR and GUITAR's test case template, taking human interaction (events) into consideration. All of the manually created tests passed, on all applications, thereby supporting our previous statement.

TABLE V: Summary of metrics acquired during the case study. **G** - GUITAR, **VGT** - VGT GUITAR, **TC** - Test case

Metric	Rachota	JEdit	JabRef
Total # of generated TCs	353954	297568	110164
Test case length	4	3	2
Total # of applicable TCs (G/VGT)	1000 / 53.5	1000 / 1	1000/ 8.5
Failed TCs applicable in both tools (G/VGT)	4% / 100%	0% / 100%	91.5% / 100%
Failed GUITAR test cases (1000 test cases)	50.1%	16.95%	28.3%
Average TC exe. time (G/VGT)	10.65s / 31.74s	12s / 32s	19.143s / 31.875
Stand. dev. TC exe. time (G/VGT)	1.71s / 3.049s	0.8325 / 0	0.377s / 0.353s

However, as shown in Table V, the component-based approach also failed on average in 31.78 percent of the 1000 test cases generated by GUITAR. Further, GUITAR failed on average 31.83 percent of the time for the test cases also applicable for the VGT approach, i.e. out of the 53.5, 1 and 8.5 test cases respectively for Rachota, JEdit and JabRef. Analysis of the test results for the failing test cases showed that there

were three types of failure conditions:

1) Interactions with disabled components;
2) Interactions with null components; and
3) Timeouts.

Root cause analysis of these failures showed that they were primarily dependent on the AUT's initial state during execution. For instance, JEdit is a text editor with the majority of its GUI components related to text manipulation and editing. However, if no document is loaded into the application these components are disabled. GUITAR's ripper could capture the components and create test cases with them but during execution the test cases failed because the components' disabled state. This problem is caused by GUITAR's EFG implementation that, as stated, does not include component state information. However, this design decision presents a potential threat to the validity of the study's results since the AUT could have been initiated with an example document that had enabled more GUI components and perceivably raised the test success-rate. Likewise, if state information had been taken into account, it is perceived that the test suites success-rate had been higher overall. However, since VGT GUITAR was executed in the same context, i.e. with disabled buttons, the EFG implementation choice is not perceived to affect the result validity for the comparison between the two approaches.

Additionally, it was observed that the test execution for the VGT approach was significantly slower (a factor of three) than for the component-based approach. This result also supports the result from the experiment, where VGT was also slower. The identified cause was once again the implementation of the VGT GUITAR prototype, as explained in Section IV-A.

Consequently, our hypothesis that the VGT GUITAR prototype is not currently applicable in practice is supported. One factor of this conclusion is the immaturity of the prototype but also the lack of domain knowledge taken into account during test case generation, which also affects the component-based approach, as shown by the high failure rate for disabled components. This result also shows the importance of placing the AUT in a suitable state before ripping and executing the test cases. In addition these results presents a need, and area of future work, for smarter test suite generation algorithms to mitigate false test results due to invalid test scenarios, i.e. scenarios that do not take user interaction with the AUT into account. Specifically, this functionality is required for a hybrid tool since ripping with the component-based approach will result in test cases that instrument components visible in the GUI model but not necessarily on the rendered GUI, e.g. menu items or components outside the area of view.

Thus, we state that our hypothesis that GUITAR is significantly more applicable in practice than the VGT GUITAR prototype is supported. Especially if the AUT's initial state is considered such that disabled GUI components are not present, etc. However, it must be noted that whilst the VGT prototype performs both input and output assertions of the AUT's behavior, GUITAR only asserts that input can be performed and that the AUT does not report an exception. GUITAR can also assert AUT state files but these assertions are still limited since they do not evaluate that the rendered GUI state actually conforms to the state file model of the GUI. Thus providing further support for the need of a tool that combines the component-based and VGT approaches to facilitate both automated system and acceptance testing.

V. Discussion

Our study shows that in terms of false test results the component-based approach is more suitable for system testing whilst the VGT approach is more suitable for acceptance testing. We therefore posit that a combination of the two is the most suitable.

Previous work [4] has shown that test execution with VGT suffers from robustness problems related to the approach use of image recognition. However, image recognition provides VGT with flexibility and allows for simple creation of powerful input/output oracles, but as a drawback the image recognition is slow. In contrast, the component-based approach is associated with quick, but also robust, test execution. However, the approach requires that the used tool has access to the AUT's GUI library, which makes it inflexible and restricts the approach applicability for certain systems, e.g. distributed systems or systems written in several programming languages. Furthermore, component-based oracle creation is a challenge due to the required technical knowledge about the AUT and is despite such knowledge only able to assert the GUI model rather than what is actually rendered on the screen [19].

These properties support our statement regarding the suitability of combining the two approaches since it would allow for robust and fast test execution with flexible oracles that could assert both the GUI model as well as the graphical output shown to the user. To the authors' best knowledge, Hewlett Packard's (HP) tool Unified Functional Testing (UFT) [14] is the only tool available with this multi-approach functionality. The industrial applicability and how well this functionality supports multi-approach scripts is however unknown.

Another important implication of our results is that component-based tools should not be used for acceptance testing, due to the large number of false negative results. Whilst false positives may increase development cost due to unnecessary root-cause analysis of working code, false negatives can result in lingering defects in the delivered system. Thus stressing the need for industrial practitioners to evaluate the capabilities of their test tools/approaches in relation to what purpose they are used for. In addition, this result implies that using only one test method for quality assurance is not enough but rather that testing needs to be performed on several, or perhaps all, levels of system abstraction, from component level to GUI level, with different tools/approaches. This statement is partially supported by Berner et al. [20]. However, further work is required to evaluate the criticality of our statement and we do not suggest that automation is the only solution, e.g. acceptance tests could/should be performed manually to explore the space of potential defects in the AUT. However, due to recent trends in software industry towards more continuous integration/development [21], a need

exists that warrants future work regarding fully automated GUI based testing tools. In addition, this need also warrants the development of a completely GUI based version of the GUITAR tool, i.e. a continuation of VGT GUITAR. The reason is because, as discussed in this work, it is not possible to obtain test coverage with the component-based approach for systems where AUT access is restricted, e.g. distributed systems and systems developed in multiple languages.

The developed prototype of VGT GUITAR does not fulfill this need, since, as shown in this work, the prototype is currently not applicable in practice. However, the study does showcase that such a tool could be developed and that it has potential use in practice, as shown in the experiment. Further support for the development of such a tool is given by the related work into Random Visual GUI Testing [9] in which random inputs and random test case execution was used in combination with image recognition to test a real-world web-application. However, in contrast to related work, this paper is pivotal in showing how automated model creation, test case generation, and image recognition can be combined for GUI-based testing.

A. Threats to Validity

The first threat to the validity of the presented results is the use of a small application for the experiment that could affect the external validity of the results. However, as discussed in Section III, since GUI level testing abstracts the system's logic into a black-box the results are perceived general for all GUI based software. A greater threat is instead the choice of instances for each of the 18 defined mutation operators, i.e. the instances could have been developed in different ways that perceivably could have affected the results. However, this does not affect the validity of the given set, which encompasses a larger set of real application defects. For instance, for Mutation Operator 1, removal of a GUI component from the GUI, often occurs in practice and also entails cases where the GUI component is changed. In the latter case, GUI change, we have redundant evaluation since the experiment evaluates several types of GUI component change, e.g. changing the appearance, type and GUI library of the components. As such, the threats to external validity are considered low.

Another threat concerns the case study and the choice of tested applications. The choice is motivated by the applications previous use in academic research and due to their range of functionality, graphical appearance and means of interaction. These applications are therefore perceived to provide results of sufficient external validity. Furthermore, because of their diversity they add to the internal validity of the results but given that other applications could have been chosen there is a minor threat to the construct validity related to the research design. However, since results from the experiment support the results from the case study this threat is also considered minor, i.e. the overall construct validity of this work is considered adequate.

Third, there is a threat to the external validity of the case study in terms of general applicability. We clearly state that we evaluate the applicability of the tools in practice, which is not to be confused with industrial applicability. Due to the open source nature of the applications and limited data set we make no broader claim of the current applicability of the approaches. However, for GUITAR, previous research does support the tool's general applicability, whilst for VGT GUITAR the results show that the tool is not applicable and therefore requires future work. Such future work should include also analysis of the costs associated with implementation and general use of the tool in industrial practice, including the cost of analysis and identification of false test results.

Lastly, the results of this work indicate, but do not show, that a combination of the two test approaches would improve GUI test efficiency. However, since this factor was not explicitly evaluated in the study we only claim that the reported results support our hypothesis but does not provide evidence for it. Identifying further support for the hypothesis is therefore another subject of future work warranted by the perceived synergy effects between the different approaches presented in this work.

VI. Supporting and Related Work

Mutation testing is a practice where developer mistakes are seeded as intentional faults in a software in order to evaluate the quality of a test technique or tool and thereby show that said tests are adequate to find defects in the system [8], [22]. The technique appeared in the early 70s and has, as shown by Jia and Harman, been applied in many different areas of testing for a plethora of programming languages [8]. However, to the authors' best knowledge, our work is the first where the concepts of mutation testing are applied on a GUI level of system abstraction in order to test GUI based test cases.

GUITAR is a second generation tool that rips, generates and replays test cases automatically for GUI based testing [1]. The ripping procedure interacts with the AUT and records events and properties of the GUI components in the AUT that are then used to create an event-flow graph [23]. This graph represents the possible interactions that can be performed on the AUT's GUI and is used to generate test cases, which can be replayed for automated GUI-level regression testing [22], [24]. As such, the tool adheres to the model-based testing (MBT) paradigm. Thus, removing the need for costly manual test case development, maintenance and execution. However, due to the tool's reliance on the component-based GUI testing approach, its applicability is restricted to Java and Python AUTs.

Visual GUI Testing is referred to as third generation GUI based testing and is a tool driven technique, with tools such as Sikuli [16] and JAutomate [15], which uses image recognition for interaction and assertion of an AUT's behavior. The benefits of the technique is its flexibility of use for any GUI based system but because of its immaturity it is also associated with robustness problems, i.e. false test results are reported due to image recognition failure [4].

Previous work has evaluated the benefits and drawbacks of the component-based approach and VGT approach for web-systems [6], but to the authors' best knowledge there is no

research that evaluates the false test result rate or potential benefits of combining the two techniques or compared their applicability on desktop systems.

VII. Conclusions

In this work we have performed a comparison of component-based and VGT based GUI level testing through an experiment and a case study. Results from the experiment show, with statistical significance, that the component-based approach reports more false negatives than VGT for acceptance tests but that VGT reports more false positives than the component-based approach for system tests. This result relates to how the approaches interact with the AUT. Whilst the component-based approach interacts and asserts the GUI model rather than the pictorial GUI, the approach interactions with the AUT differs from human interaction, which makes the approach suitable for system testing but inapplicable for acceptance testing. In contrast, VGT interacts with the AUT in the same way as a human user, i.e. through the pictorial GUI, which makes it applicable for both automated system and acceptance testing. However, since the test cases need to take user interaction into account, system test cases become more cumbersome to create than for the component-based approach. Further, this infers a reliance not just on the behavior of the AUT for the VGT test cases to pass, but also that the appearance of the AUT's GUI is correct. Consequently, tools used for acceptance testing must operate on the same level of abstraction as the intended user, i.e. pictorial GUI level for GUI driven applications.

Further, a case study presented in this work shows that GUITAR is applicable in practice, but that the VGT based GUITAR prototype tool still requires future research and development. More explicitly, better test case filtering and bitmap image capture algorithms are required to raise the percentage of VGT applicable test cases. In addition, state space explosion must be mitigated during test case generation to make it possible to generate test cases that are more representative of industrial grade test cases for both approaches.

Consequently, this work indicates that there are complementary benefits and drawbacks with component-based testing and VGT that infers that a combination of the two approaches would be the most suitable in practice. Future work is therefore warranted into tools that can perform automated GUI based testing with both approaches as well as purely VGT based testing to fulfill the software industry's need for test automation to support continuous integration, development and deployment.

References

[1] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014.

[2] E. Borjesson and R. Feldt, "Automated system testing using visual gui testing tools: A comparative study in industry," in *Proceedings of the 5^{th} IEEE International Conference on Software Testing, Verification and Validation (ICST 2012)*, Montreal, Canada, 2012, pp. 350–359.

[3] E. Alegroth, R. Feldt, and H. Olsson, "Transitioning manual system test suites to automated testing: An industrial case study," in *Proceedings of the 6^{th} IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, Luxembourg, 2013, pp. 56–65.

[4] E. Alégroth, R. Feldt, and L. Ryrholm, "Visual gui testing in practice: challenges, problemsand limitations," *Empirical Software Engineering*, pp. 1–51, 2014.

[5] E. Alégroth, "On the industrial applicability of visual gui testing," Department of Computer Science and Engineering, Software Engineering (Chalmers), Chalmers University of Technology, Goteborg, Tech. Rep., 2013.

[6] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Visual vs. dom-based web locators: An empirical study," in *Web Engineering*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8541, pp. 322–340.

[7] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[9] E. Alégroth, "Random visual gui testing: Proof of concept," *Proceedings of the 25^{th} International Conference on Software Engineering & Knowledge Engineering (SEKE 2013)*, pp. 178–184, 2013.

[10] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] C. Lowell and J. Stell-Smith, "Successful automation of gui driven acceptance testing," in *Proceedings of the 4^{th} International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 03)*, Berlin, Heidelberg, 2003, pp. 331–333.

[12] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[13] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[14] T. Lalwani, M. Garg, C. Burmaan, and A. Arora, *UFT/QTP Interview Unplugged: And I Thought I Knew UFT!*, 2nd ed. KnowledgeInbox, 2013.

[15] E. Alégroth., M. Nass, and H. Olsson, "JAutomate: A tool for system- and acceptance-test automation," in *Proceedings of the 6^{th} Software Testing, Verification and Validation (ICST 2013)*, Luxembourg, 2013, pp. 439–446.

[16] T. Chang, T. Yeh, and R. Miller, "Gui testing using computer vision," in *Proceedings of the 28^{th} international Conference on Human factors in Computing Systems (CHI 2010)*, Atlanta, US, 2010, pp. 1535–1544.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[18] B. N. Nguyen and A. Memon, "An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 216–234, 2014.

[19] R. A. P. Oliveira, A. Memon, V. N. Gil, F. L. S. Nunes, and M. Delamaro, "An extensible framework to implement test oracle for non-testable programs," in *Proceedings of the 26^{th} International Conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, Vancouver, Canada, 2014, pp. 199–204.

[20] S. Berner, R. Weber, and R. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th International Conference on Software Engineering 2005 (ICSE 2005)*, St. Louis, USA, 2005, pp. 571–579.

[21] M. Fowler and M. Foemmel. (2006, May) Continuous integration. Access: Octber 2014. [Online]. Available: http://www.martinfowler.com/articles/continuousIntegration.html

[22] P. R. Mateo, M. P. Usaola, and J. Offutt, "Mutation at system and functional levels," in *Proceedings of the 3^{rd} IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2010)*, Paris, France, 2010, pp. 110–119.

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

[24] A. Memon and M. L. Soffa, "Regression testing of GUIs," in *ESEC/FSE-11: Proceedings of the 9^{th} European software engineering conference held jointly with 11^{th} ACM SIGSOFT international FSE*, New York, USA, 2003, pp. 118–127.