On the Industrial Applicability of TextTest: An Empirical Case Study

Emil Alégroth
Chalmers University of Technology
Department of Computer Science
and Engineering
SE-412 96 Göteborg, Sweden
emil.alegroth@chalmers.se

Geoffrey Bache
Evry AB
Olof Asklunds gata 10
42130 Västra Frölunda, Sweden
geoff.bache@gmail.com

Emily Bache
Bache Consulting
Flunsäsliden 25
SE-418 71 Göteborg, Sweden
emily@bacheconsulting.com

***Abstract*—Software systems are becoming more complex, not least in their Graphical User Interfaces (GUIs), which presents challenges for existing testing practices. Pressure to reduce time to market leaves less time for manual testing and increases the importance of test automation. Previous research has identified several generations of automated GUI-based test approaches with different cost-benefit tradeoffs. Whilst test automation provides fast quality feedback it can be associated with high costs and inability to identify defects not explicitly anticipated by the test designer.**

TextTest is a capture-replay tool for GUI-based testing with a novel approach that overcomes several of the challenges experienced with previous approaches. Firstly the tool supports Approval Testing, an approach where ASCII-art representations of the GUI's visual state are used to verify correct application behavior at the system level. Secondly it records and replays test scripts in a user defined domain specific language (DSL) that is readable by all stakeholders.

In this paper we present a three phase industrial case study that aims to identify TextTest's applicability in industrial practice. The paper reports that the tool is associated with (1) low script development costs due to recording functionality, (2) low maintenance costs, on average 7 minutes per test case, (3) better defect finding ability than manual system testing, (4) high test case execution performance (In this case 500 test cases in 20 minutes), (5) high script readability due to DSL defined scripts, and (6) test suites that are robust to change (In this case 93 percent per iteration). However, the tool requires a higher degree of technical skill for customization work, test maintainers need skills in designing regular expressions and the tool's applicability is currently restricted to Java and Python based applications.

***Keywords*-TextTest, System Testing, Industrial Study, Approval Testing**

I. Introduction

GUI based software testing is a practice of growing importance in the software industry, applied both for system and acceptance testing [1]. Testing can be performed both manually (scenario-based or exploratory) and/or automated with one out of three generations of approaches: coordinate-based (1^{st} generation), component-based (2^{nd} generation) or Visual GUI Testing (3^{rd} generation)) [2]. However, despite growing commonality in practice, automated GUI-based testing approaches have different applicability challenges. For instance, the 1^{st} generation is sensitive to GUI change, the 2^{nd} generation is dependent on GUI model access and the 3^{rd} is associated with lack of robustness [2]. Furthermore, all approaches are associated with non-negligible development, execution and/or maintenance costs [1], [3]–[8]. Costs that can provide positive return on investment over time but still warrant continued work into alternative test solutions that focus on mitigating these costs [2], [9].

TextTest [10]–[12] is an open-source tool with a novel approach that is perceived to solve several of the above stated challenges. The novelty of the tool lies in its use of a "Describer" that transforms the pictorial GUI shown to the user into ASCII-art. ASCII-art is in this context an abstract representation of the GUI drawn with alphanumerical symbols in contrast to a bitmap image drawn with pixels. The ASCII-art captures the GUI state as an abstract screenshot for each GUI event during record and replay of a test case. Assertions are then made by using textual comparison to compare the expected result, the "gold standard" from the test recording, to the new ASCII-art captured after replay of the test case. Consequently asserting all state changes/events triggered by the test case implicitly. The tool can therefore identify system level defects regardless of where or how they manifest in the GUI and regardless if they are explicitly or implicitly caused by the user scenario. Combined with a domain specific language (DSL) paradigm for script creation, system test cases can be created that are both powerful and readable by any stakeholder. In addition, the textual representation of both input (scripts) and output (ASCII-art) lead to high maintainability since multiple input/output artifacts can be maintained simultaneously by applying the changes made to an artifact to all other artifacts affected by the change with regular expressions.

Previous iterations of TextTest have been published in related work [10]–[12], including how to use the tool for specification by example based (SbE) testing [13]. However, to the authors' best knowledge, only limited empirical evidence has been provided for the tool's applicability in practice [10], [12]. In this paper we present a three phase industrial case

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

study at CompanyX[1] that explores the applicability, but also longer term feasibility, of TextTest in practice, including costs, benefits and drawbacks [14]. The high-level contributions of this work to the general body of knowledge on automated GUI based testing are therefore:

1) A description of TextTest, its approach and its benefits and drawbacks for automated GUI based testing.
2) Empirical support for the applicability and feasibility of the approach in industrial practice.

II. Related work

Industrial GUI-based testing tools generally support capture/record and replay functionality to mitigate development and maintenance costs [1], [3], [15]. However, automated scripts still require the user to manually create assertions of correct Application Under Test (AUT) behavior based on oracles [15]. These oracles define the expected results and can be composed of AUT properties, GUI components, GUI bitmaps, etc., on different levels of detail granularity. A study by Qing and Memon [15] showed that the type of used oracle has direct impact on the defect finding ability for GUI-based testing. As such, this work extends their work by presenting a new type of oracle based on ASCII-art for implicit assertions of the entire system state with defect-finding ability regardless of how or where a defect manifests on the GUI. We further present this concept in Section III.

Another solution to capturing defects regardless of how they manifest is test case generation, e.g. driven by models, with tools such as GUITAR [16]. However, the generated test cases are assertion-based and large test suites are therefore required to cover every GUI state, which becomes time consuming to execute and thereby has feasibility issues in practice.

Three generations of GUI-based test techniques have been defined [2]. TextTest is a 2^{nd} generation tool and therefore uses component/widget information, accessed through hooks into the AUT's GUI library, for interaction [2], [6]. This approach is associated with robust test execution and robustness to GUI change. However, this approach inherently limits the 2^{nd} generation tools' applicability to the programming languages they have built-in support for.

The purpose of test automation, on any level of AUT abstraction, is to provide faster AUT quality feedback through frequent regression testing. Automated regression testing on lower levels of system abstraction, e.g. component level with unit tests [1], is common practice. However, due to lack of robust and cost-effective tool support for higher level tests, e.g. system and acceptance tests, automated GUI level testing is less common. Higher level tests are instead performed with manual, but also tedious, costly and error-prone, test practices [1], [8], [17].

A. TextTest background

TextTest was initially presented for acceptance-test driven development (ATDD) in 2003 [12] after a study at Carmen Systems. The tool operated by sending/reading data from the AUT's technical interfaces to stimulate/assert AUT behavior where any discrepancy (barring process ID, timestamps, etc) were classified as failures. However, this process required the AUT to produce output that was, or could be converted to, meaningful plain text. Results from the study showed that the "time-to-green-bar", i.e. time to passed test, could be kept low with costs similar to unit-testing.

In 2004, Andersson and Bache extended the tool with record and replay functionality for use-case scenario based GUI testing [11], i.e. system tests. This functionality (a library known as 'PyUseCase') allowed developers to create scripts in a domain specific language (DSL) that makes the tests readable, definable and verifiable by the AUT's stakeholders.

In 2005, the tool was extended with thread event listeners that support automated synchronization between the scripts and the AUT, which lowers development cost since manual synchronization is a core challenge with most GUI-based techniques/tools [18]. The tool was then evaluated in three projects at Carmen Systems, which concluded that script parallelism is key to achieve enough performance for continuous integration. Parallelism that is achieved in TextTest through creation of independent scripts and/or by mocking out time consuming components, i.e. replacing components with static components or values.

III. GUI-based testing with ASCII-art

PyUseCase is replaced in the version of TextTest evaluated in this study with a tool called StoryText with better AUT behavior recording. StoryText uses component-based (2^{nd} generation) GUI test technology to hook into the AUT's GUI library to create ASCII representations of the AUT's GUI. This allows an approach we now refer to as Approval Testing that in contrast to an assertion-based approach does not require the user to record or create explicit assertions to verify the AUT's behavior during test creation. We formally define Approval Testing as "*the assertion of an application under test's correct behavior based on comparison of state information from scenario-based stimulation with previously approved application state information*". In TextTest, Approval Testing means taking the user's tacit approval of the correctness of the GUI they see on the screen as the basis for implicit assertions that are recorded automatically during test creation. In essence, the tool records assertions that require everything the user sees on the screen to stay the same between two versions of a system, albeit in a low-fidelity textual representation.

During test recording, TextTest records the system's GUI state information in ASCII-art and stores it in a file. The ASCII-art is created by a "Describer" - a code artifact that hooks into the AUT's underlying GUI library, and renders each visible component as text. During replay of a test case, the Describer creates a new ASCII-art file that TextTest uses to assert correct AUT behavior by performing a textual comparison between the two generated ASCII-art files. This approach is effective in terms of execution time compared to, for instance, bitmap comparison but also more maintainable since text

[1]The company's name has been anonymized for confidentiality reasons.

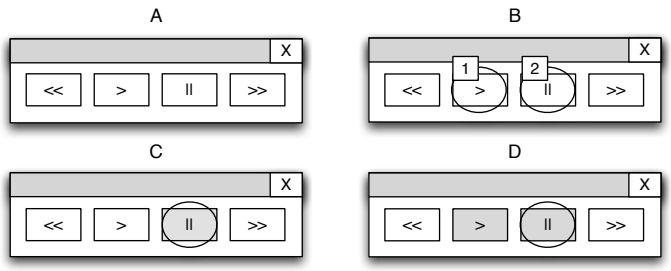

Fig. 1. The GUI (A) is to be tested with a scenario where the play button is first pressed followed by the pause button (B). An assertion is then made to check that the pause button is lit (C), i.e. the GUI's correct behavior. However, assuming that the final state is as shown in (D), asserting that the pause button is lit will result in a false negative result, since it will not detect the AUT defect - the play button is also lit.

files can be updated more easily than bitmaps. However, for AUTs with custom GUI components the Describer requires customized code to understand how to render said components in ASCII. The Describer can also be calibrated to render more or less ASCII-art detail, which effectively affects the defect-finding ability. As such, the fidelity of the ASCII-art compared to the pictorial GUI must be considered to mitigate false test results. Finding a suitable fidelity level can be a challenge and requires a person with intricate knowledge of TextTest, the AUT and its domain.

When the textual comparison identifies differences, these are presented to the user who determines if the differences are indicative of an AUT defect or if they are expected/intended due to changes to the AUT. That is, if maintenance is required of the script, AUT GUI code or AUT code logic. Consequently, TextTest will report defects on any level of AUT abstraction that manifest through the GUI, i.e. system defects.

Approval testing can thereby identify defective AUT behavior regardless of how or where it manifests on the GUI. In contrast, tools that rely on explicit assertions are limited to finding defects within the test scenario. Explicit assertions can therefore miss defects that cause propagation of faulty AUT behavior, i.e. false negative results, as visualized in Figure 1.

However, the use of ASCII-art for Approval Testing also presents some challenges. First, since the Describer requires hooks into the GUI library of the AUT it restricts TextTest use to certain programming languages. Explicitly, AUT's written in Java and Python with the Swing, Eclipse, Python TK and GTK GUI libraries. This is a limitation that TextTest inherits from the reliance on 2^{nd} generation GUI testing technology [18].

Second, constructing filters for the ASCII art with regular expressions can be a challenge and can be error-prone. Such errors can cause output to be compared incorrectly, leading to false test results. TextTest users therefore need skills in designing regular expressions, which for advanced regular expressions is a higher technical knowledge requirement than for some other GUI based test techniques, e.g. VGT [2].

A. Scripting with TextTest

Test script development in TextTest is performed according to the following six steps.

1) TextTest is started,
2) The user presses the "Record" button in TextTest, which starts StoryText and the AUT
3) The user records a test scenario,
4) The user is prompted to designate domain specific terms for each of the performed AUT interactions,
5) The user approves the test script, (or rejects it, in which case they may start again from the beginning)
6) The test suite is updated with the new test script and the corresponding ASCII-art file.

In the fourth step of this process, the user defines the domain specific language (DSL) for the script's events, which are automatically associated with interactions that the replayer can understand within a TextTest artifact called a user interface (UI) map [13]. The UI map serves as a middle-layer that connects interactions with GUI components, identified through component properties such as ID or label, to user defined actions written in natural language. As an example, the UI map could include a translation of the formal action "*click <button = "start">*", to the DSL notation "*Start processing*". Whilst the recorder and replayer can understand the formal action's syntax, as it is stated, this syntax lacks domain knowledge of the action's purpose. This purpose should be given by the DSL notation to make the test script readable and understandable by a stakeholder, e.g. user or customer, regardless of her technical knowledge.

DSL defined actions can also be grouped to create more advanced events, which further promotes the creation of consistent and understandable test cases at a higher level of abstraction. The GUI components captured during recording of an action, or group of actions, can also be changed into variables that makes it possible to apply the DSL action(s) on any GUI component in the AUT that can be tested using said action(s). Furthermore, every time a new script is recorded, the recorder will try to link the formal actions of the new script to already specified DSL notations, mitigating the need for manual "refactoring" of scripts. As such, the user incrementally creates a custom scripting language with embedded domain knowledge tailored to the AUT being tested. TextTest puts no restrictions on the DSL notation the user defines but it is recommended that the user defines a notation suitable to capture the AUT's requirements in a manner similar to a use case, aligning the test to the AUT's requirements.

Once a holistic DSL has been defined, the tool can also be used for Specification by Example (SbE), Behavior-Driven Development (BDD) or Acceptance Test-Driven Development (ATDD) styled testing where the script is written before the functionality of the SUT has been implemented [13]. The test cases then serve both as requirements and system tests of the AUT's conformance to said requirements. However, in contrast to most SBE, BDD or ATDD techniques and tools, TextTest supplies this capability at a GUI level of abstraction.

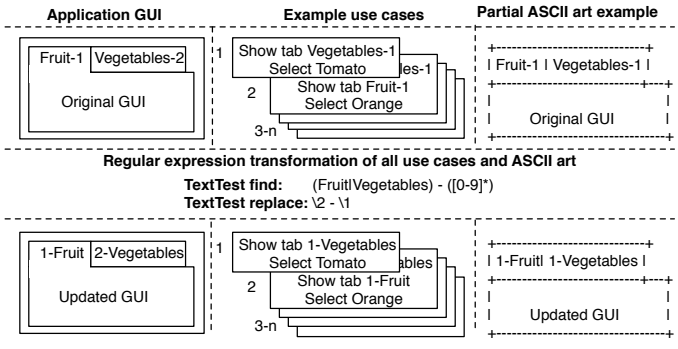

Fig. 2. Visualization how a regular expression can be used to update the use cases and ASCII art after changes to the GUI. Note that the ASCII art only shows the AUT's start screen, in the tool these renderings would show all states of the GUI associated to one use case.

B. TextTest maintenance

To support parallel test case execution, as stated in Section II-A, all test cases recorded with TextTest start the AUT directly from the command line at the start of the test and terminate the AUT at the end to achieve test case independence. As such, test case independence helps lower test suite execution time but has the drawback that changes to an AUT event will cause all test cases dependent on said event to fail and require maintenance. Parallel test execution of GUI tests is technically supported in TextTest by running the tests against a virtual display, e.g. Xvfb [19].

When a large number of test cases all fail together due to a single cause, that is, either a defect in the AUT or an expected change, TextTest often does not require the user to maintain every failed script individually, in contrast to most other test techniques [1]. Due to the textual representation of both scripts and output, TextTest is able to aggregate similar changes and present them to the user in a grouped list to help shorten root-cause analysis time. Using the information from the aggregated list, TextTest test artifacts can then be maintained by applying the maintenance solution of one artifact to all artifacts that share its behavior through the use of regular expressions.

A visualization of an example scenario where the user would use this maintenance functionality is shown in Figure 2. In the example the names of tabs in an AUT GUI, shown in the top of the figure, has been updated, shown in the bottom of the figure, which affects both the DSL defined scripts and the associated ASCII-art. Since the tabs are used in several scripts, they all require maintenance. By applying the regular expression presented in the middle of Figure 2 all TextTest artifacts are sub-sequently maintained. In this way all conflicts in the test suite associated with the changes of the GUI are resolved simultaneously.

When test failures are caused by several intertwined changes to the AUT, which require several regular expressions to update the "gold standard" ASCII-art files, the user can apply regular expressions one at a time, and rerun the textual comparison step in between. This can be done without re-running the entire test suite, which is an advantage since re-computing the text comparisons is generally much faster than re-running all the test cases.

If the user is unsure of the nature of the defect and can't immediately identify a good regular expression to apply, they may want to reproduce the defective state and examine it in the actual application GUI. This can be done by following the test case steps manually and either analyzing the screenshots generated by the tool or the AUT's response, or by running the test step-by-step and examine each step individually in the AUT.

As such, script maintenance in TextTest is fast, given that it is performed by a user knowledgable in writing regular expressions. For advanced regular expressions, this user knowledge may be a challenge since incorrect regular expressions can degrade the test suite. A recommended best practice is therefore to version control all artifacts developed in TextTest.

Further, in order to sustain a working test suite over a longer period of time, the user, as stated, must also maintain the Describer. All standard GUI components from the Java Swing, Eclipse, Python TK and Python GTK libraries are known to the Describer but for custom components special code is required. As new functionalities are added to the AUT it may therefore be required to add new, or calibrate, Describer functionality and update existing golden standard ASCII renderings accordingly. Describer maintenance, although relatively infrequent compared to Test maintenance, requires expert knowledge of the tool and the AUT and is therefore perceived as a core challenge for the applicability of TextTest in general practice.

IV. Industrial Case Study

In order to identify support for the industrial applicability and feasibility of TextTest, an empirical case study [14] was performed at CompanyX. This section will present the characteristics of the company and the case study methodology.

A. Company description

CompanyX is a software development company with approximately 3000 employees around the world, of which 350, with over 30 different nationalities, work at the location where the study was performed. The company's product portfolio includes over 40 tracks of products for aviation and railway management. One of the main products is a rich-client, GUI driven, application for the aviation industry with a complex back-end that performs resource optimization.

TextTest is used throughout the company, both for system level testing through technical interfaces and through the GUI. These testing capabilities coupled with the low development and maintenance costs of TextTest scripts are the main reasons why the tool is used in CompanyX. In addition, the tool has partly been developed within the studied company and integrated into the company's test process.

In the project where the study was performed, 30 developers, in 3 cross-functional Scrum teams, work together to formulate requirements, components and tests. The AUT for the project is written in the Java programming language and has in the order of hundreds of thousands lines of code.

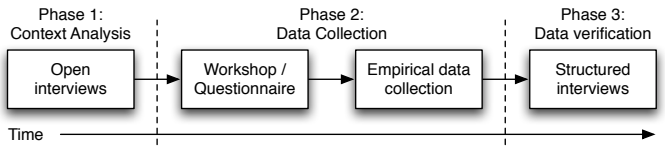

Fig. 3. Visualization of the research methodology used during the case study at CompanyX.

TextTest is integrated into the project's development process where verification is performed with 600 TextTest test cases, 700 unit test cases and 1000 manual scenario-based test cases. Development is performed in two-week sprints at the end of which 48 man-hours of manual regression testing is performed in a release test with a subset of the manual test cases. The automated tests are executed during every new build for continuous integration, starting with the unit tests, followed by a new build, followed by the TextTest test cases. Testing is decentralized within the organization, meaning that both testers and developers work with tests. However, for the more technically advanced testing, e.g. TextTest, test experts are primarily responsible. There are three TextTest experts in the project that each day cycle the responsibility of running and maintaining the TextTest test cases whilst the others work with other testing activities. Lastly, TextTest test cases are based on the AUT's requirements to support traceability between them.

As such, CompanyX is representative of companies of medium to large size where safety-concerned, but perhaps not safety-critical, software systems are being developed with agile software methodology with cross-functional teams and decentralized test activities.

B. Research design

The case study was divided into three phases, as visualized in Figure 3, guided by the research questions:

- *RQ1: Is TextTest an effective GUI testing tool?*
- *RQ2: Is TextTest a cost-effective GUI testing tool?*

Phase one was performed as an exploratory pre-study with open interviews held with the person responsible for TextTest maintenance and testers at CompanyX to acquire contextual information about the company, processes used and how said processes integrate TextTest. The open interviews were guided by both of the study's research questions as well as a set of sub-questions to get a shallow, yet holistic view, of the use of TextTest at CompanyX. This pre-study was conducted over two full work-days at the company, i.e. 16 hours, and also aimed to identify subjects and projects for the continued data collection, i.e. snowball sampling. Factors of interest for the sampling included the subjects'/projects' knowledge and experience/relevance with/for TextTest use, generalizability of projects' characteristics and the subjects' ability to contribute with empirical data to answer the research questions.

Phase two of the study was divided into two parts, both with the purpose of exploring the applicability and feasibility of TextTest in practice (*RQ1 and RQ2*). Part one was performed during a full-day workshop attended by 16 employees from CompanyX, chosen through convenient sampling, which were introduced to the tool through a series of exercises of varying difficulty. The exercises were designed by the person responsible for TextTest maintenance at CompanyX and required the use of all core functionality to create, maintain, and execute TextTest scripts. After the workshop, a questionnaire was electronically distributed to the attendees with ten questions with a mix of forced choice questions, open ended questions and 10 point Likert scale questions. Nine out of the 16 attendees responded the questionnaire (Response rate 56.2 percent). The questionnaire aimed to elicit background information, e.g. industrial experience, of the attendees as well as their perceptions about the tool based on their experiences from the workshop and previous use of the tool. Perception was in the questionnaire triangulated through questions regarding the perceived value, enjoyment and difficulty of working with the tool. Analysis of the questionnaire results showed that the sample group had sufficient distribution of different roles, e.g. developers and testers, but was biased towards practitioners with years of industrial experience. A questionnaire was chosen in favor of interviews due to resource constraints. Measured metrics of particular interest to answer RQ1 in this phase were:

- *Perceived tool value:* Required for the tool to be applicable in practice
- *Enjoyment:* Perceived easier to apply if enjoyable to use
- *Difficulty of use:* On different levels of AUT abstraction and in comparison to other tools/techniques

In the second part of phase two, a six week long longitudinal study was performed in a live project at the company chosen based on the results from phase one. The purpose of this phase was to collect quantitative results to evaluate TextTest's long-term applicability and feasibility (*RQ 1 and 2*). In addition, this part of the study aimed to compare the use of TextTest to the manual scenario-based test practice used in the project. Manual scenario-based testing was chosen as a benchmark for the comparison since it was, out of the available test practices at the company, the most mature. Furthermore, the purpose of both TextTest scripts and the manual testing is system level testing with test cases of similar complexity, appearance and GUI interaction, which make them comparable. The data collection was performed by industrial practitioners working in the project, following a data collection protocol that was specified prior to the study. 10 test iterations were measured by three different testers in the project to mitigate data collection bias. Further, the testers had no stake in the project, which is also perceived to help mitigate data collection bias. The data protocol was documented in a text document in parallel with the test execution and maintenance but no reviews were held to ensure data validity. However, the testers were asked after the study if they had experienced any problems during the data collection and no problems were reported. The practitioners also perceived the data set to be representative for normal working conditions at CompanyX. After the study, the text document with the synthesized results was sent to the leading

#	Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	Median	Std. Dev.
1	On a scale from 1-10 how VALUABLE do you find TextTest as a testing tool (1. Worthless, 10.Very valuable)?	7	8	5	10	6	8	9	8	6	8	1.59
2	On a scale from 1 to 10 how FUN did you find TextTest as a tool (1. Not fun, 10. Very fun)?	7	6	2	8	2	8	8	6	6	6	2.37
3	If you had the choice, would you use TextTest for testing in your project (1.Yes, 2.maybe, 3.no)?	2	1	2	1	2	1	1	1	2	1	0.53

TABLE I
SUMMARY OF KEY QUANTITATIVE DATA COLLECTED DURING THE WORKSHOP AND THE MEDIAN VALUES EVALUATED FROM SAID VALUES.

researcher who performed the data analysis. Measured metrics of particular interest to support RQ1 and answer RQ2 in this phase were:

- *Number of failed test cases:* To be compared to defects found (False positives) (RQ1)
- *Number of defects found:* Required for the tool to be applicable (RQ1)
- *Maintenance time:* Required low to be feasible (RQ2)
- *Challenges with tool:* If few, it supports the applicability and feasibility of the tool (RQ1 and RQ2)

In phase three, the collected data from phase one and two were analyzed and then verified through triangulation through semi-structured interviews with two out of three testers that performed the data collection in the second part of phase two. Thus, providing supporting information regarding the applicability and feasibility of TextTest in practice based also on a longer perspective than the six weeks of the study (*RQ 1 and 2*). The semi-structured interviews were performed with an interview guide consisting of 41 interview questions that were composed from the analysis results. For instance, the qualitative results from phase one and the quantitative results from phase two indicated the feasibility of TextTest in practice and therefore the tester's perception of this result was elicited. Thereby providing a third data source of triangulation. Similar questions regarding perception or knowledge were asked for all key concepts that would support or contradict previous results to answer the study's research questions. Each of the interviews were recorded and then transcribed.

The collected data, together with results of a review of previous work on TextTest, was then synthesized through qualitative analysis and descriptive statistics. Formal statistics were not used because the collected dataset from part two in phase two, despite the data collection protocol, was incomplete and because the number of data points were not sufficient. In order to raise the construct and internal validity of the results we have therefore chosen to include key results as raw data in the paper. Qualitative observations have also been supported by quotes from the interviews. For the reader's convenience we have presented the median and standard deviation to some key questions from the questionnaire from the workshop in phase two. However, since these medians are calculated based on quantification of nominal scales they are only to be considered directional markers of what direction the respondents were leaning overall.

C. Workshop Results

Key results from the questionnaire survey from the first part of phase two have been summarized in Table I. The workshop attendees had on average 9.1 years of industrial experience and had different roles at the company, including test designers, system analysts, developers and tech leads. As such indicative of a suitable sample for the study but with a bias towards more experienced practitioners. Further, several of the participants answered that they had previous experience with TextTest. However, analysis showed that this experience was quite low, on average 60 hours in total per person.

The respondents where asked to name and rank the testing techniques that they had previously worked with in a professional capacity. The ranking was done on a five point nominal scale from easy to difficult in an open question, i.e. without any suggested tools or techniques for the participants to choose from. During analysis the respondents individual answers were grouped into four categories. These categories were manual testing, TextTest testing, xUnit testing and "other". Manual testing in this context includes both scenario-based and exploratory testing, whilst xUnit includes all types of unit testing regardless of language. TextTest refers to both GUI and non-GUI based testing with the tool since the workshop included exercises for both. Finally "other" testing techniques refer to practices/tools that were only mentioned by single respondents, e.g. Fitnesse testing or JMeter testing. The purpose of this question was to get more detailed information about the participants previous test experience but also to see if there were any correlation between their previous experience and their perceptions about TextTest. Analysis of previous experience contra perception revealed no consistent patterns. Analysis did however show that the participants ranked xUnit to be the easiest technique to work with, followed by manual testing, then TextTest and finally other types of testing. It should be noted that these techniques are used for different test purposes and are therefore not inherently comparable, but the result does indicate the users' perception towards ease of use of the different tools/techniques.

It was hypothesized prior to the survey that the tool would be perceived difficult because of the required use of regu-

lar expressions for filtering and maintenance, as explained in Section III-B. An explicit question was therefore asked regarding this difficulty; on a scale from 1 to 10 where 1 was very hard and 10 was very easy. The median value of the respondents answers was 5 with a standard deviation of 1.12. Hence, just below the center value, which infers that normal regular expressions are only moderately difficult.

To identify other aspects of the tool that were perceived challenging, the participants were asked an open ended question to rank what they found the most difficult with the tool, or usage of the tool. Analysis of the results was inconclusive because of the variety of answers, e.g. some thought GUI testing with ASCII-art was the most difficult, other's that creating mockups was the most difficult. This shows that there are challenges with the tool related to factors not captured by the survey, e.g. the user's subjective perceptions towards tooling. This result does however lets use hypothesize that the tool has a higher learning threshold.

Table I shows the respondents answers to how valuable and fun they found the tool was to use overall. Value is defined as value in terms of defect finding ability compared to other test practices and fun as the enjoyment of using the tool for system testing. The enjoyment factor was evaluated because it may sway the decision to use to the tool in a project and because it perceivably reflects the tool's usability and learnability, where a complex tool is perceived as less enjoyable. The median perceived *value* of the tool was considered high, i.e. 8 out of 10, whilst the participants experience of how *fun* the tool was to work was just above the middle, i.e. 6 out of 10. We therefore conclude that the tool is perceived valuable but only slightly considered fun to work with.

Finally, the participants were asked if they would introduce TextTest in their own projects, individual results shown on line 3 in Table I. The question was divided into a yes, maybe, no answer on which the respondents responded on median 1 based on the quantified answers from the questionnaire where yes was quantified as 1, maybe to 2 and no 3. Thus, the participants want to use the tool in their projects.

Note, quantification of nominal scales only show directional markers of the samples perceptions. Further, despite the participants' average previous TextTest experience, these results are to be interpreted as first impressions of the tool.

D. Longitudinal study Results

Table II and Figure 4 presents the quantitative results that were collected during the longitudinal study in part two of phase two. The table shows the number of failed test cases per subsystem of the project per test iteration out of the estimated 600 test cases available for the system. An iteration is defined as the time between one successful test execution to the next, including re-runs and required maintenance efforts. The table also shows the number of new defects that were found per test iteration and the amount of time spent on maintenance in each iteration to bring the test suite up to reasonable quality. 46 test cases failed on average per test suite per iteration but only one of these failed test cases, on average, was due to a defect in the AUT. As such, the ratio between false positive test results and actual defects is quite high. However, since TextTest aggregates the test result for quicker test failure root cause identification, as stated in Section III, the percentage of failed test cases that identified actual defects are greater than 2 percent (1 defect / 46 failed tests) in practice.

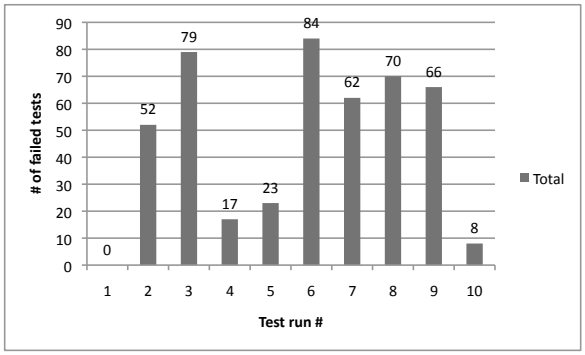

Fig. 4. Number of failed test cases reported during the longitudinal study. The bars shows the number of failed test cases per test iteration.

#	Date	SS1	SS2	SS3	Total	# New defects	Time (hours)
1	10/7/13	-	-	-	-	1	5
2	10/8/13	8	44	0	52	1	2.5
3	10/14/13	0	65	14	79	3	6
4	10/15/13	0	9	8	17	1	5
5	10/17/13	0	23	0	23	0	-
6	10/21/13	0	23	61	84	0	-
7	10/24/13	0	23	62	62	1	5
8	10/30/13	22	0	47	70	0	6.25
9	10/31/13	1	1	42	66	1	3
10	11/6/13	0	8	0	8	0	11
	Average	3.1	19.6	23.4	46.1	0.8	5.469
	Std. dev.	7.094	21.219	26.504	31.146	0.919	2.592

TABLE II
SUMMARY OF THE QUANTITATIVE DATA ON FAILED TEST CASES COLLECTED DURING PHASE 2. THE DATA HAS BEEN DISPLAYED GRAPHICALLY IN FIGURE 4. **SSX** - SUB SYSTEM X.

Furthermore, as shown by Figure 4, the number of failed test cases fluctuated over time with no observable pattern. The reason for this result is because the test failures are dependent on which AUT functionality required maintainance. As such, it is natural that the number of tests that require maintenance also fluctuate over time. Another reason why no pattern could be identified is the short study period of six weeks, resulting in only ten test iterations. Over longer periods of time it is expected that a pattern would be seen that less test cases fail as the quality of the AUT improves over time. As such, the results indicate that there is a degree of sensitivity in the scripts, reflecting the need for both the AUT's behavior and GUI appearance to be correct for the test cases to pass. Thus supporting the tool's use for system testing. Furthermore, the average of 46 test cases per iteration represent a failure rate,

on average, of 7 percent since there were an estimate of 600 test cases in the test suite. Thus providing the test suite with an estimated average robustness to change of 93 percent.

Furthermore, as presented in Section III-B and supported by the data in Table II, the maintenance costs of the test suites are low, between 2.5 to 11 hours per iteration with an average of 5.5 hours for all suites. As such, the average maintenance cost is roughly 7 minutes per failed test case, which is supported by an interview result from phase one where it was reported that 300 test cases from a suite of 500 test cases was maintained in 16 hours, i.e. with an average time of roughly 3 minutes per test case. The difference in maintenance time reported from the interview and the study could be because of several factors, but the most plausible is due to dependencies between the failing test cases, i.e. that there were more dependencies between the failing test cases in the case reported in the interview. As such, maintenance became quicker since, as reported in Section III-B, the maintenance of one test can be applied to repair more test cases within the test suite.

Additionally, interview results from phase one identified that the execution time of a test suite of 500 test cases is only 20 minutes because of parallel execution of 50 to 100 test cases at once at CompanyX. The low execution time allows the test suite, if required, to be executed every build to provide continuous quality assurance feedback to the developers on a system level of abstraction. Coupled with the automated unit testing, the two techniques provides a holistic view of the current quality of the AUT and its subsystems.

E. Interview Results

First, the interviewees were asked about how their test related tasks were divided across a normal work week. They reported that they spent 40-50 percent of their time with exploratory testing, 30-40 percent on TextTest test case maintenance, 10-20 percent on manual scenario-based testing and lastly 10-20 percent on team related work, e.g. helping developers with their test related activities. As such the required maintenance of TextTest test cases is significant but the testers also reported that the tool has beneficial defect-finding capabilities. "*It finds defects that cannot be found using exploratory testing for instance, e.g. non-deterministic test cases.*" The defect finding ability is supported by the quantitative results that show that TextTest finds on average one defect per test iteration whilst the manual scenario based testing finds less than one per iteration. This is also supported by the results from part one of phase two where the workshop participants ranked the perceived value of the tool as high.

Further, the interviewees reported that the learnability of the tool is high and that anyone regardless of background can learn the tool because of its record and replay paradigm. "*Yes, it (learnability) is not dependent on the user's technical knowledge since tests are recorded and work against the GUI.*" This statement infers that even though first impressions of working with the tool, as measured in the workshop, may be that the tool is more difficult than other tools at first glance, the tool has high learnability overall. However, regarding enjoyment, the interviewees' responses support the results from the workshop, i.e. that it is moderately fun to work with. "*It's easy but I wouldn't say fun. It was more fun in the beginning, but you learn everything quite quickly and then it's not fun anymore, but easy.*" The testers ranked exploratory testing and QTP/UFT to be the more fun than TextTest and scenario-based testing to be the least fun. They also reported exploratory testing and TextTest to be the most efficient at finding defects, followed by QTP/UFT and finally manual scenario-based testing. As such, the manual scenario-based testing was considered both the least fun and least valuable.

The interviewees learned TextTest by working with the tool but proposed that new testers should start by observing a TextTest expert working with the tool. "*...if a new employee starts (using TextTest), it is enough to sit next to someone experienced for 3 days and learn.*" The reason is the novelty of the tool's approach that requires a different mindset from other automated test tools. Also, how to handle the aggregated failures reported by the tool to identify the root cause of failures quickly and maintain the test case(s) if required.

The tool's main benefit was however reported to be the speed of test execution. This provides support for continuous quality assurance and integration and give both testers and developers frequent feedback that instills trust in the system's quality and stability. Second, it also allows the test suites to be run many times in order to identify non-deterministic failures. '*You can run a test case 1000-10.000 times and it goes really quickly. You can see if the failures are deterministic, etc.*" Hence, failures can be isolated that would not be cost effective to identify using manual test practices.

The testers reported that they had trust in the tool, i.e. that they did not perceive it to report many false positives or negatives. "*Well, I trust TextTest, it can fail but it's the nature of the tool if you have added things (to the AUT)*". This statement supports the quantitative data analysis that showed that the test suite had an estimated robustness of 93 percent. However, a contributing factor to the interviewees statement of trust in the tool is that the person responsible for tool maintenance works closely with the testers and provide them with tailored support if an issue with the tool arises. "*It (Feasible TextTest testing) is reliant on the maintenance person being here*".

The interviewees' also reported some drawbacks with the tool. First, the tool's use of 2^{nd} generation GUI based test technology that differs from end user interaction with the system. "*You crawl under the hood of the system sometimes... The tool uses system events for waits.*" This limits the tools applicability for end user scenarios, i.e. acceptance testing. The benefit, as reported, with this approach is the tools ability to effectively synchronize script execution with the AUT automatically through thread observation. Thus limiting the required time spent on manual maintenance to ensure synchronization. Another reported drawback is the ASCII-art renderings since they are more abstract than the bitmap GUI. "*TextTest draws the GUI which makes it more abstract*".

Consequently, the interviews supported the previously collected results, showing that TextTest is both an applicable and

Tech.	# TCs at CompX	hours / week	Defects found / run	TCs / hour	TS exe. time	TCs / day
Text-Test	600+	10-32	1+	75-375	0.3 hours (20 min)	600-3000
MSB tests	1000+	8-16	$\leq$1	1.125	533 hours	3

TABLE III
SUMMARY OF THE APPROXIMATIVE QUANTITATIVE VALUES GATHERED DURING THE STUDY RELATED TO MANUAL SCENARIO-BASED AND TEXTTEST TESTING. **TECH** - TECHNIQUE, **TC** - TEST CASE, **TS** - TEST SUITE, **MSB** - MANUAL SCENARIO BASED, **COMPX** - COMPANYX.

feasible tool in practice.

F. Compared to Manual Scenario-based Testing

10-20 percent of the testers' time is spent on manual scenario-based testing each week at CompanyX, i.e. 30 test scenarios every two weeks, equal to 8 to 16 hours of manual testing per tester per test iteration. In parallel, each tester spends 10 to 32 hours maintaining the TextTest test cases. Consequently, twice as much time is often spent on script maintenance compared to the manual scenario-based testing.

However, whilst roughly 600 TextTest test cases can be executed every build, i.e. often at least once every day, only 90 manual test case scenarios are performed every two weeks. The studied project does however have in excess of 1000 GUI-based manual test scenarios, which with the current practice for manual testing has a total execution time of 22 weeks. Consequently, whilst 150 TextTest test cases can be executed in average per hour (assuming two builds per day), only 1.125 manual scenario-based test cases can be executed. Furthermore, the TextTest tests were reported to regularly find regression defects and was ranked as one of the top techniques in finding defects by the interviewees, whilst manual scenario-based testing was ranked the least effective. These results are summarized in Table III.

V. DISCUSSION

Market demands for faster time-to-market put requirements on software development companies to use more agile practices and strive towards continuous integration, development and delivery. In this context, the importance of automated testing becomes ever more evident for all levels of system abstraction. Thus, warranting more research and development into new tools and techniques, like TextTest, especially since previous research has presented several limitations, e.g. high cost and lack of robustness, and a need for further research into automated GUI level testing [1]–[8].

The results from this work, in conjunction with related work, show that TextTest is a suitable tool in practice with several beneficial properties. The main benefits of the tool, support shown in parentheses, are (1) *low script development costs due to recording* (Related work and Part 1 of Phase 2), (2) *low maintenance costs provided by the textual representation of input and output* (Part 2 of Phase 2), (3) *its defect finding ability provided by the Approval testing approach* (Phase 2 and Phase 3), (4) *high test script execution performance supported by parallelized script execution* (Related work, Phase 2 and 3), and (5) *the readability of scripts because of the DSL based scripting that in turn supports SBE, BDD and ATDD* (Related work, Phase 2 and 3) [13]. Further, (6) results from the study showed that the test suites in the studied context had an average robustness of 93 percent, i.e. 93 percent of the test cases were unaffected by AUT change (Part 2 of Phase 2). Out of the seven percent of failing test case, 2 percent continuously identify AUT defects each test iteration, which is greater, in average, than the manual test practices in the studied context. Results 1, 3, 4, 5 thereby answer research question 1, whilst 2 and 6 provides an answer for research question 2.

The implications of these results are that the tool can provide quick return on investment but also instill trust in the quality of the product being delivered. Especially since the study shows that the defect finding ability of the tool is greater than manual test practices and can identify infrequent failures that would not be feasible to identify manually. Consequently, TextTest is both a complement and replacement to existing tools, techniques and practice that enable companies to comply to market needs for faster time-to-market.

The main drawbacks are that the tool requires hooks into the AUT's GUI library which means TextTest is currently restricted to Java and Python. The tool also requires a high degree of technical knowledge to calibrate and customize, primarily, the tool's Describer. Normal maintenance work requires skill with regular expressions. These drawbacks are challenges for the general applicability of the tool but what the impact of said drawbacks are is still a subject of future research. However, since the impact is perceived as non-negligible, future research and development of the tool aims to mitigate these challenges to further improve the tool's general applicability in practice.

This work also provides an academic contribution in showing that Approval testing, with ASCII-art textual comparison, is a viable approach in practice, thereby providing support for a new approach within GUI based testing that warrants future research. Approval testing's ability to identify failures regardless of how they manifest in the GUI is an advantage over assertion driven testing and therefore yet another subject that warrants further research, e.g. how to apply Approval testing to other test paradigms. The use of a domain-specific language in a capture/replay tool also warrants further study, to examine the specific effect of this practice on efficiency of test creation and maintenance. Further, this work provides a general contribution that higher level system testing can be conducted with alternate approaches, warranting further work into novel techniques and tools that can support the software markets growing needs for test automation.

A. Threats to validity

Construct validity: With the exception of the workshop in part one of phase two, this research was performed in an industrial project environment. Thus providing empirical data from the actual use of TextTest in practice with high construct validity. Additionally, the workshop results are considered to have sufficient construct validity because it was performed with industrial practitioners that were intended users of the tool that had both technical and domain knowledge.

Internal validity: All phases of the study provided close to consistent empirical evidence to answer the research questions, i.e. data triangulation and method triangulation could be performed. For instance, the applicability of the tool was from the workshop and interview results identified as high, supported by the quantitative results that showed how the tool can consistently identify regression defects on a GUI level. Feasibility was triangulated in similar manner, where quantitative results show low maintenance costs supported by the interview results regarding the tool's feasibility. As such, the internal validity of the qualitative results are perceived sufficient to answer the study's research questions. However, due to the small number of quantitative data points, future work is required to also verify our results with formal statistics.

External validity: The study was only performed at CompanyX, which restricts the generalizability of the presented results to companies with similar characteristics, i.e. larger software systems with the applicable GUI libraries, cross-functional agile teams, dedicated TextTest maintenance personnel, etc. The company has been described to the highest level of detail we were able due to a confidentiality agreement, but we perceive this description to be sufficient to capture the key characteristics required for other contexts. However, further studies in more companies are still required to broaden the results of this work. Further, due to TextTest approval testing approach it is perceived that the measured results are not valid for other automation tools.

Reliability: The research methodology has been described to a level of detail such that we are confident that the study should be replicable in the given context or similar contexts [14].

VI. Conclusions

In this paper we have presented TextTest, a GUI based test tool with a novel approach to system testing built on textual comparison and Approval testing of ASCII rendered representations of the AUT's pictorial GUI. Related work together with a three phase study presented in this paper shows the tool's industrial applicability. In the first phase of the case study a pre-study with open interviews was performed, followed in phase two of a workshop with 16 attendees from the studied company, CompanyX, as well as a longitudinal study were costs of using TextTest in practice were collected. Finally, semi-structured interviews were performed to triangulate the study's results and results from related work on TextTest.

The study results show that the tool is both applicable and feasible in industrial practice due to low development and maintenance costs, support for approval testing, high script execution performance, high script readability and robust test scripts. However, the tool requires technical knowledge to maintain the tool's ASCII-art describer, currently only provides support for Java and Python systems and requires dedicated TextTest maintenance personnel in practice.

Consequently, TextTest is applicable in industrial practice but associated with some challenges that warrants future work to improve/extend the tool and its approach.

References

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

[2] E. Alégroth, "On the industrial applicability of visual gui testing," in *On the Industrial Applicability of Visual GUI Testing*. Chalmers University of Technology, 2013.

[3] D. Rafi, K. Moses, K. Petersen, and M. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Automation of Software Test (AST), 2012 7th International Workshop on*, june 2012, pp. 36–42.

[4] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI testing tools using accessibility technologies," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009, pp. 243–250.

[5] E. Horowitz and Z. Singhera, "Graphical user interface testing," *Technical eport Us C-C S-93-5*, vol. 4, no. 8, 1993.

[6] E. Sjösten-Andersson and L. Pareto, "Costs and Benefits of Structure-aware Capture/Replay tools," *SERPS'06*, p. 3, 2006.

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

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

[9] J. J. Gutiérrez, M. J. Escalona, M. Mejías, and J. Torres, "Generation of test cases from functional requirements. A survey," in *4ş Workshop on System Testing and Validation*, 2006.

[10] J. Andersson, G. Bache, and C. Verdoes, "Web applications, multithreading, parallel testing and multiple components: Further adventures in acceptance testing."

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

[12] J. Andersson, G. Bache, and P. Sutton, "Xp with acceptance-test driven development: A rewrite project for a resource optimization system," in *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003, pp. 180–188.

[13] E. Bache and G. Bache, "Specification by example with gui tests-how could that work?" in *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2014, pp. 320–326.

[14] 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.

[15] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, p. 4, 2007.

[16] 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.

[17] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 408–418.

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

[19] D. Bolcsfoldi, D. Mandelbaum, and P. Truter, "Apparatus and method for control of multiple displays from a single virtual frame buffer," Oct. 18 2010, uS Patent App. 12/906,933.