Beyond Plain Video Recording of GUI Tests

Linking Test Case Instructions with Visual Response Documentation

Raphael Pham, Helge Holzmann, Kurt Schneider
Software Engineering Group
Leibniz Universität Hannover
Hanover, Germany
{raphael.pham, kurt.schneider}@inf.uni-hannover.de
helge.holzmann@se.uni-hannover.de

Christian Brüggemann
Application Lifecycle Service Center
Capgemini Deutschland GmbH
Hanover, Germany
christian.brueggemann@capgemini.com

Abstract—**Information systems with sophisticated graphical user interfaces are still difficult to test and debug. As a detailed and reproducible report of test case execution is essential, we advocate the documentation of test case execution on several levels. We present an approach to video-based documentation of automated GUI testing that is linked to the test execution procedure. Viewing currently executed test case instructions alongside actual onscreen responses of the application under test facilitates understanding of the failure. This approach is tailored to the challenges of automated GUI testing and debugging with respect to technical and usability aspects. Screen recording is optimized for speed and memory consumption while all relevant details are captured. Additional browsing capabilities for easier debugging are introduced. Our concepts are evaluated by a working implementation, a series of performance measurements during a technical experiment, and industrial experience from 370 real-world test cases carried out in a large software company.**

Keywords-component; Automated Test; Graphical User Interface; Video; Code Tracing

I. INTRODUCTION: AUTOMATED GUI TESTING

Despite development of different approaches, GUI testing remains a difficult task. Typically, a GUI supports user input by mouse and keyboard, thus the theoretical execution possibilities of a GUI explodes compared to the case of a method of discrete values. Regression testing of test-suites has long become an integral part of Quality Management and Maintenance. Test-driven development calls for automated regression testing after each change made to the code [2]. Also, changes in the environment demand repetition of test suites. Manually repeating long test suites takes too much time and effort and will not be performed in practice. Therefore, automation of test execution is indispensible for repeating long and complex suites of test cases.

Zimmermann et al. [18] analyzed what made a good and usable bug report from the perspective of the bug fixing engineer. They found “steps to reproduce” to be the most desired information. Video recording could provide a usable impression of test case execution. Still the video can barely give insight to steps necessary to reproduce the failure as only the response behavior is documented. When GUIs are tested automatically, the following aspects need to be captured and conveyed to the developers responsible for debugging:

- The deviation of required and observed behavior of GUI elements must be recorded. A video with sufficient coverage of the screen and resolution in time is an option to capture enough detail for analysis during debugging.
- The test case step that caused the failure should be identified and documented, too.

As generic screen videos will not qualify for this purpose, we propose a tailored video-based documentation that is linked to the executed test case instructions.

In Section II, we discuss specific challenges in debugging GUIs. Those challenges are compared to related approaches in Section III. We propose concepts to meet the challenges and present our approach of tailored video-documentation in Section IV. It was developed in cooperation with Capgemini, a leading IT consulting company which is present in more than 30 countries with over 115,000 employees. The resulting demonstrator was evaluated with respect to technical quality, such as performance and memory consumption (Section V). It was then applied to 370 real industrial GUI test cases at Capgemini. In Section IV, we consider limitations of our approach. We discuss our findings in Section VI and conclude.

II. CHALLENGES IN DEBUGGING GUIS

One of the main challenges in GUI testing is often neglected: trouble shooting and debugging. For the sake of software quality and in the interest of saving resources, findings from the GUI testing process and related tools must feed seamlessly into debugging [19]. Myers describes this debugging task as two single steps [1]: the first step is to detect the defect in the code of the application under test (AUT). The second step is to fix it. Failure refers to deviant behavior of the AUT compared to the required behavior during test execution, while defect refers to flawed lines of code in the AUT which result in the failure [3]. To detect a defect, unexpected behavior needs to be analyzed. One challenge often encountered with GUI tests is that failures (deviations) are not immediately uncovered.

The example dialog in Fig. 1 requires the user to type in her first name in text field A and her last name in text field B. Furthermore, the user shall be able to use TAB to switch from text field A to text field B. The used test framework identifies dialog elements via ID. One GUI test case could be:

1. open dialog, use ID to select text field A
2. send keyboard events in order to type “First”
3. send TAB keystroke, switch to text field B
4. send keyboard events in order to type “Last”
5. find Button “ok” by ID and click it
6. check if text field A says “First”
7. check if text field B says “Last”

Supposing that the TAB function is not implemented properly and step 3 causes the dialog to lose focus. In step 4, the send keyboard events are lost and do not appear onscreen. Only test step 7 reveals the deviation. Neither a screenshot at the point of time when the deviation is revealed (step 7) nor a video recording of the whole test case properly documents the failure: The screenshot only shows the dialog with an empty text field B. While this information is also included in a video, the video would not show the loss of keyboard events, as they did not appear onscreen. For debugging, this is critical information. However, the video could help in revealing step 3 as the deviation point (nothing is happening after text field A has lost the focus in step 3). Still, consulting the test case instructions and comparing the observed behavior is essential in order to gain understanding of the failure.

The case of correcting wrong or outdated GUI test cases presents a similar situation: Supposing that in step 1 the ID to select text field A is outdated. Failure would be registered at step 6. Again, video and screenshot lack critical information and would benefit from further semantic information. In this case, the inclusion of executed test case instructions could facilitate understanding of the expected behavior[1].

We advocate video recording of GUI tests and wish to enhance its usefulness from the perspective of the debugging engineer. Through a video recording the engineer gains a comprehensive insight into the test procedure and the responses of the AUT. As these examples show, this information should be enriched with insight to the intended test procedure in order to facilitate debugging.

We propose to document both the AUT’s reaction onscreen as well as the current status of the test case (executed test case instruction) and present these views of the test procedure in a synchronized manner. This would reveal the critical test case step earlier and facilitate understanding of the failure. Thus, the search for the deviation is shortened and the engineer gains insight in the test procedure.

[1] In our survey (section 5, User Reactions) GUI testers at Capgemini reported the increase of OS-version of Windows would regularly bring changes of IDs for standard GUI elements of system dialogs. This in turn renders several test cases wrong and results in the repeating task of correcting already passed test cases in order to maintain regression testing.

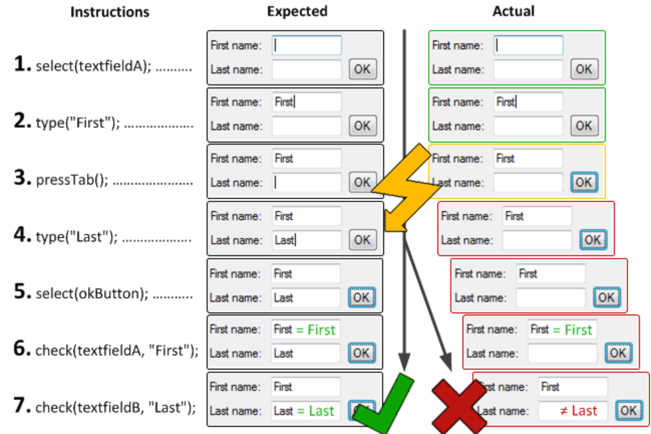

Fig. 1. Deviation of actual behaviour from required behaviour.

Rather sophisticated mechanisms are required to support debugging. A specialized screen recorder is needed as well as a code tracing utility. Main challenges for implementing such a tailored video-documentation are (1) the very fast speed of automated operations in GUI tests and (2) tracing of test code, which describes the operations to be performed. The fast operation speed requires taking a lot of screenshots to capture all performed actions without overlooking a potentially important change. On the other hand, memory has to be managed in a very efficient way. Hence, storing of duplicate screens has to be avoided. How to solve the other challenge (tracing of the test code) depends on how the test code will be handled by the test framework. If it is written in a scripting language, it will be interpreted and evaluated directly by the framework. Since the operations will be performed by the framework, tracing can easily be integrated. Other test frameworks like Ranorex [4] do not offer such a scripting language and provide a library to develop the test code in a full-fledged programming language like C#.

III. Related Work

We know of no tool in the domain of GUI testing that offers the functionality to fully solve the described challenges of synchronized screen recording with simultaneous test code tracing. Many existing screen recorders like Camtasia Studio by TechSmith [5] provide very good screen capture capabilities, but for more universal purposes than the special case of GUI testing. Their main purpose is the recording of the screen for tutorials or demonstrations of applications. They have to work in an efficient way, no matter what type of content appears on the screen. Therefore, they have to treat and handle full screen actions and very fast animations like videos as efficient as usual applications with graphical user interfaces.

Other screen recorders consider the operation of GUI applications by reacting to user inputs to take snapshots. DebugMode’s Wink is a working example [6]. However, an intensive evaluation of this tool has shown that its method of event-driven recording is not suitable for capturing GUI tests. Interesting actions in GUI tests are triggered by input, but do not appear immediately. It is a difficult task to

determine how long the internal action resulting from the input lasts and how long the screen has to be recorded after each input. Common test frameworks wait until an event occurs rather than for a pre-defined timeframe.

The approach of the solution presented in this paper is to react to screen changes by mirror graphic drivers. This concept is often used by virtual network computing software (VNC). The open source project TightVNC software [7] uses a mirror driver to detect changes on the screen and transfers it efficiently via network to the controlling client. Very similar to it is UltraVNC [8]. The developers of this solution created an own mirror driver and they also developed a screen recorder using the driver [9]. They have extended the open source CamStudio by RenderSoft [10].

Unfortunately, all these solutions are stand-alone applications. They have been developed for screen recording and creating videos of it, but it is difficult to integrate these solutions into another application. For this purpose, Microsoft offers the Expression Encoder Pro [11] with a specialized API to realize screen capturing in custom developed applications. Besides the actual screen recording, timestamps of every captured screen frame are needed in order to synchronize these frames with the timestamp information of the traced test code instruction. We know of no solution that provides these data in its recorded videos.

The only complete third-party solution which allows both of the requested tasks (screen capture and code tracing) is the test framework, which is integrated in the Microsoft Team Foundation Server [12]. It uses the mentioned Expression Encoder to record automated GUI tests and Microsoft's historical debugger IntelliTrace [13] to simultaneous trace the executed test code that operates the application under test. Personal communication with Microsoft revealed that synchronizing and connecting video and traced code is possible. However, this only applies to manually executed GUI tests and does not apply to automated regression testing.

IV. Concepts of Tailored Video-Documentation

A. *Screen Capture*

Obviously, a main task in video-documentation of automated GUI tests is recording the screen. It is less obvious at which point of time the screen should be recorded, and whether it may be sufficient to capture only a part of it. For debugging, frequent and complete screenshots are preferable. However, this leads to huge amounts of data being recorded and stored – a challenge to the speed of recording and the memory used for mid- and long-term storage. The intention is to take as many shots as necessary, but also to manage the memory in an efficient way.

Time-driven and event-driven capturing of screenshots is common in different approaches, used by the several screen recorders (e.g. DebugMode's Wink [6]). In *time-driven approaches*, the whole screen is captured after a pre-defined interval. In theory, this approach can catch all actions appearing on the screen, given the interval has been chosen short enough. Several frames per second may be necessary. In practice, however, the time it takes to capture and save a snapshot of the screen limits the frequency of screenshots. At a lower frequency some activities on the screen may not be captured. This may cause problems in debugging when important intermediate states have not been recorded and therefore cannot be considered during analysis. In addition, time-driven approaches also take redundant snapshots at the same rate when nothing changes on the screen.

The *event-driven approach* captures the screen only when a defined event occurs. Examples for such events include keyboard or mouse inputs. However, not every action visible on the screen is a direct consequence of such an input event. Displayed information and screen layout may change during simulation or time-consuming computations, as well as due to background processes. Therefore, not all visible actions are covered and captured in an event-driven approach. For example, Wink [6] recorded the press of a button but not the resulting opening of a window after some computation time.

In *code-driven capturing*, the internal event of execution of a test code instruction is used to trigger snapshots. Similar to the above-mentioned user input events, not every action onscreen is triggered by such an internal event. In addition, only some of the instructions result in a visible action. Therefore, this approach requires a lot of space for redundant snapshots, despite rather low coverage.

As time-driven, code-driven and user-input-event-driven approaches have drawbacks regarding onscreen change coverage or space consumption, we instigated other event-driven options. To capture a sufficient amount of the actions that result in visible screen changes, it would be most suitable to consider those visible screen changes as events and react directly when they occur. We call this approach *output-driven screen capturing*. As operating systems like Windows do not provide native support to detect screen changes we employed a so-called mirror driver. It works like a usual graphics driver, but without producing any visible output. When an output to the screen occurs, the mirror driver will be informed and stores the coordinates of the changed rectangle. As a result, memory can be saved by capturing just that changed area instead of the whole screen on every change. Fig. 2 illustrates this principle.

The shaded rectangles on the left represent the changed segments. As shown in step 4, more than one change can occur at a time. However, saving all of these changed areas separately takes more time than saving one larger area. It also takes more memory space because it cannot be compressed as much. Hence, a common bounding box is computed around all changes, detected at one time. That bounding box also contains areas of the screen that have not been changed, but missing a change would be much worse than wasting a little space.

After capturing the segments, they are stored in a hash map (middle column in Fig. 2). The hash map keys are specialized objects which identify the corresponding segments by the hash of its bytes. On conflicts of the hash values an exact comparison will be performed byte by byte. This allows detecting duplicates. Due to the nature of GUI tests, many views, buttons and other controls appear repeatedly in a test. Thus, they result in the same segments with the same hash values and do not need to be stored. This

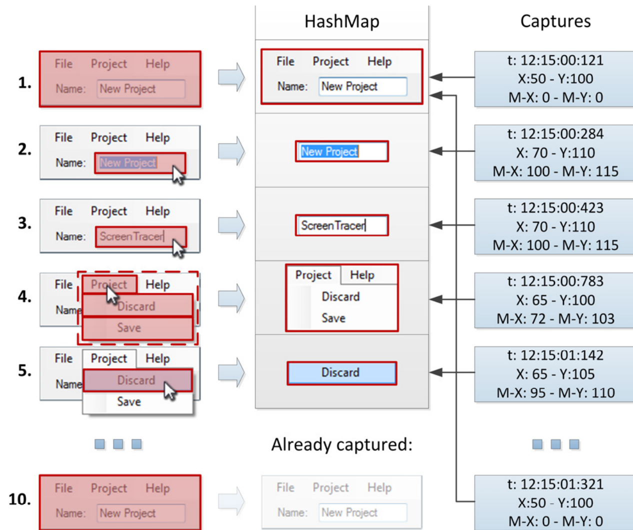

Fig. 2. Output-driven capturing of changed screen segments.

can be accelerated by compressing the segments with PNG or any other graphic compression format.

Finally, the coordinates of every segment have to be stored. That is the list on the right in Fig. 2. Due to the duplicate detection mentioned above, multiple captures in this list may point to the same segment. They also contain the mouse coordinates, because the mouse cursor is not contained in the segments. The cursor must be reconstructed on viewing the recorded video.

B. *Code Tracing*

In order to analyze and understand failures, the desired behavior must be known. It is then compared to the observed behavior. By connecting the test code to the captured screen, GUI changes are linked to their semantics. In order to relate executed test instructions with actions on the screen, the test code must be traced. This code tracing has to log every test code instruction with the timestamp of its execution. Based on this information the exact test process can be reconstructed. For this purpose, the test code has to be instrumented with specific trace instructions. This depends on the test framework used as well as on the interpretation policy of test case instructions. Pre-compiled and independently running test code is more difficult to trace than code interpreted by the test framework. Generally, code-injection based on concepts of aspect-oriented programming (AOP) [14] can be used to inject code-tracing instructions into the test code. In our case, the source of the test code was not available and we resorted to byte code weaving [22] to insert timestamp instructions.

C. *Viewing*

After running a video-documented test, the possibility to watch the records must be provided. Since recording times and strategies were optimized for a good balance of recording speed, required memory space and full change coverage, a specialized viewer application is needed. During

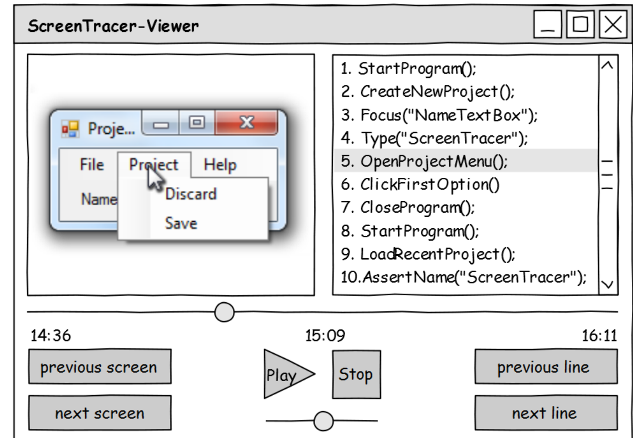

Fig. 3. Mock-up of the viewer application.

replay, the viewer synchronizes the reconstructed video with the traced test code instructions via timestamps. Thus, screen recording and tracing can work independently and do not need to communicate with each other while recording the test execution. As communication would require additional computation resources, the test execution could be affected.

The viewer's main task is to display the video alongside the documented test code. This facilitates failure analysis and debugging. The interface sketch in Fig. 3 shows the main components of such a viewer.

The time line below the screen and the code area shows the current position of the playback. The boxes above display the screen at this point of time on the left and the simultaneously executed code on the right. Control buttons ”Play” and “Stop” behave as in usual media players. A second slider below these buttons allows accelerating or slowing down the playback. This allows for quick navigation in order to find the position of the failure in the video. For diagnostic purposes the “next screen” and “previous screen” as well as “next line” and “previous line” buttons left and right offer possibilities to advance stepwise through the records. This is indispensable for navigating through the executed code. Automated tests run much faster than manual interactions. Displayed at recording speed, the video runs too fast to follow. These options allow navigating through the code like in debuggers of common IDEs.

In order to save time and memory when reconstructing the frame, so called key frames are employed. They contain the entire screen at one certain point of time during the test. This concept is similar to the intra coded frames (I-Frames) in common video coding standards like MPEG or H.264/AVC [15, 16, 20, 21]. By a statistical analysis of the segment sizes, it is often possible to create these key frames if the captured segment already contains nearly the whole screen. All other frames have to be reconstructed starting from the last available key frame. All segments captured up to the selected timestamp must be superimposed in the correct order. In this case, the key frames allow releasing all the memory needed to display previous segments as shown in Fig. 4.

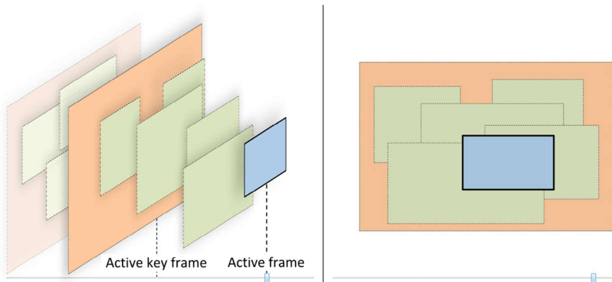

Fig. 4. Screen construction with key frames and segments.

V. Evaluation

A. Evaluation of the Operational System

Based on the mentioned deliberations and a performance evaluation (see following Section) an operational video-documentation system called ScreenTracer (ST) has been implemented (Holzmann [17], in German) according to the output-driven approach with duplicate elimination. The requirements for its practical usage were provided by Capgemini. The ST is implemented with a dynamic load-time code injection to instrument the test code. The test code is written in C# and is pre-compiled to the common intermediate language code. Thus, it runs independently, performs the test operations by itself and is executed by the common language runtime environment of Microsoft's .NET framework. The test framework is a specialized tool for a comprehensive GUI application. It monitors the tests and logs failures. It contains about 370 test cases of very different types and durations from just a few seconds to over four hours. The complete test suite of all test cases runs for more than 70 hours.

Capturing all of these tests was an intensive check of the implementation of the ST and its concepts. Additionally a specialized viewer application has been developed (see Fig. 5). This tool was given to debugging developers of Capgemini and is still employed in real debugging scenarios. Analysis of the captured data has shown that the captures on average needed less than 150 MB per hour of disk space. Due to the high number of considered test cases this amount of memory usage is representative. In the examined tests, the average size of these segments has been 635 x 305 pixels and about 40% have been duplicates. These values can also be assumed to be representative for most GUI tests.

B. Comparision of Concepts

The evaluation of the mentioned concepts has been performed on prototypes and existing software. We found user-input-event-driven and code-driven recording to provide too low coverage of recording of actions onscreen and dismissed them. We tested the user-input-event-driven approach with Wink [6] and noticed that onscreen events following a user input are missed by this approach. For the code-driven approach, we used a prototype, which took a screenshot whenever a test code instruction was executed. As

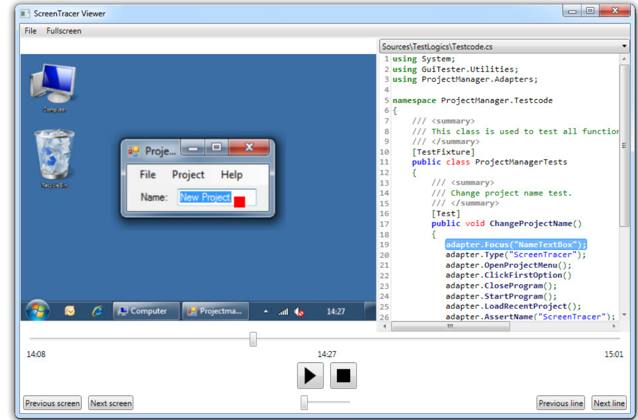

Fig. 5. Screenshot of viewer application.

test code can contain instructions, which do not result in direct onscreen change, redundant screenshots are taken. Furthermore, actions onscreen, which may occur while the test code is waiting for a specific event, are not recorded. Therefore, we concentrated our further evaluation and comparison of memory consumption on time-driven and onscreen-driven recording.

The evaluation of memory consumption was conducted on a test suite of 44 real industry test cases. These test cases include common GUI tests for opening, closing the application, saving and loading projects, as well as operation of specific GUI elements (such as traversing a tree-navigation element by keyboard). Test cases were chosen because they comprise of common operation of GUI elements: Clicking buttons, generating keyboards events, selecting elements by ID and operation of dialog elements by mouse or keyboard.

Fig. 6 shows the memory usage of the time-driven version of the ST in comparison to the output-driven version of the ST. The diagram shows a short time frame of about half an hour. The linear character of the graphs lets us assume that no different behavior in memory consumption is to be expected even if longer tests are run.

By design, the output-driven ST was triggered to take a capture every time it registered change onscreen. Its maximal frequency of screen captures was 3 frames per second. For comparison, the time-driven version took screenshots at a frequency of 3 frames per second. Due to irregular occurrence of changes onscreen the output-driven version does not capture segments at equidistant time distances (in contrast to the time-driven approach). In order to capture changes onscreen with a comparable coverage to the output-driven approach, the capturing frequency of the time-driven approach would need to be increased. This would undoubtedly result in even higher memory consumption.

As shown, the capturing of screen changes only takes a small fraction of the time-driven memory space. In this test it was about a fifth. Duplicate detection has reduced the memory usage even further.

The shapes of the graphs are quite interesting. The graph of the time-driven prototype is very straight, due to its

periodical screen capturing method. The output-driven prototype only captures the screen when it changes. This results in little waves of the graph (the memory usage for the saved segment). If nothing happens on the screen, no memory is needed and the graph is horizontal. With duplicate detection, these waves are flatter as the actions on the screen result in less segments and less space.

As shown in Fig. 6 the high consumption of resources with the time-driven approach caused a slower operation of the test suite execution, whereas the output-driven prototype finished on time. This is important, as GUI tests often wait a specified amount of time before proceeding. Slowing a test case down could result in misleading test results.

C. *User Reactions*

The ScreenTracer has been in use at Capgemini's for five months. We conducted a survey among the developers in question and interviewed them about their experiences with ST. Before ST was introduced, the test framework provided a screenshot of the GUI when the failure was detected, in some cases an event log. If the GUI test was done manually, the debugger may have been provided with a textual description of the failure. Video recording of tests was not employed by Capgemini.

User reactions were generally positive. Users reported to be able to debug faster and would recommend the tool to other colleagues. As video recording was simultaneously introduced with test code tracing, we specifically asked about the effects of gaining insights into the test code execution. Users reported to use the video for rough localization of the beginning of the departure from required behavior. Then, the presented test code is used for fine adjustments. Furthermore, test code provides semantic meaning and supports the happenings onscreen.

The following attributes were outlined:

1. Test code conveys semantic meaning:
   a. Even though the mapping of test code to screen action may be a little off, it still helps.
   b. Based on this semantic insight, wrong or outdated test cases can be corrected.
2. "The Complete Package": In some cases, the engineer was able to skip reproduction of failed test cases and began debugging earlier.
3. "It's more fun!"

VI. LIMITATIONS

After execution of a test code instruction, computation of the appropriate screen change and realization of that change takes a little amount of time. This can lead to a small delay, causing the ST to display changes and traced code out of sync. The reconstructed video of the changes may lag a little behind the presentation of the traced code, in some situations. The presentation of onscreen changes linked with test code instructions is intended to support the debugging engineer in understanding the failed test procedure. As the lagging is not great and the course of the test case execution can still be reconstructed, we consider this a minor annoyance. In time critical test cases, this may lead to

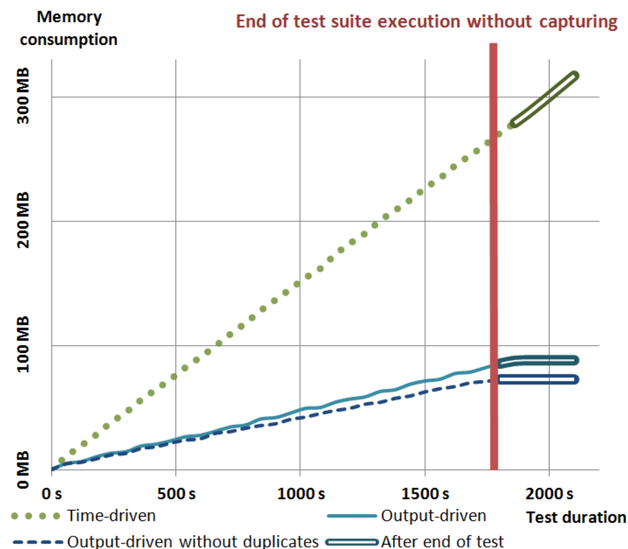

Fig. 6. Memory usage while running a representative test suite of 44 real industry test cases.

uncertainties, for example: When opening an application and closing it five times in a row, it can be hard to map each test instruction to its correct (very similar) onscreen response. Generally, the ST was not developed with highly time critical applications in mind. It is intended to establish an easy to utilize link between onscreen AUT responses and executed test case instructions.

In some cases, it may be possible for the ST to effectively replace a bothersome and laborious reproduction of a failed test case. This may be the case, when the developer already has good understanding of the AUT in question. As the ST does not (yet) provide sophisticated insight into the inner states of the AUT but only provides the external view and the intended actions, it may not replace reproduction in intensive debugging completely. Insight into the AUT's state could be gained by tracing its source code similar to tracing the test code.

Unexpected onscreen behavior, which is not directly intended by the test case, is by design documented as well in the videos of the ST, as they are also rendered by the mirror-driver. Examples are pop-ups of browser windows. This may also cause the AUT to lose focus and ultimately bring the test case to a wrong result. However, this is not a GUI test specific problem, as interferences in a test environment can occur in other test designs as well. The inspection of the video can help to identify the problem. In that case, a repetition of the test case should bring clarity.

VII. DISCUSSION

The elaborated concepts implemented in the ScreenTracer (ST) video-documentation system provide an innovative way of debugging GUI tests. They offer possibilities to trace back the behavior of the application under test as well as the complete system under test during a GUI test. Of course, each common screen recorder as mentioned in related work can capture the actions on the

screen during a test and store it in a usual video format. They can be viewed in existing media players with numerous controlling capabilities. However, none of these recorders and media players have been developed for the specialized case of capturing GUI tests. Their frame rates are often much lower, because of their universal recording purpose. Also the compression of videos is not optimized for the purpose of recording GUI tests. In GUI tests, reoccurring dialog elements (such as buttons) present the chance for compression. The ST captures only the changed segments of the screen and detects duplicates. The memory can be managed in a more efficient way, the frame rate of the recording is increased and the specialized viewer application enables advancing the captures change by change onscreen.

The code tracing is another advantage no usual screen recorder offers. Of course, separate tracing tools or logging frameworks allow the tracing of the test code. There are also specialized historical debuggers, like Microsoft's IntelliTrace [13] of the visual studio, which allows viewing the executed source code after executing it, but none of them provides the capability to view the traced test case alongside the screen capture of the system under test. The code tracer logs a timestamp to every executed line of code. This allows a link from any screen frame to the simultaneously executed line of the test code and vice versa. Thus, advancing the frame code line by code line is also possible with the viewer application like in a traditional or historical debugger.

VIII. CONCLUSIONS

Testing graphical user interfaces (GUIs) is difficult. Test cases should refer to interaction events and changes on the screen. Capturing internal events and changes of screen display is necessary to support subsequent debugging. Recording screen videos offers an opportunity.

We emphasize various reasons why standard video capturing techniques are not appropriate or sufficient for this application: They consume too much memory, take too long to capture, and still miss out relevant aspects. In addition, viewing GUI test videos raises very different requirements and demands.

We developed a number of concepts to meet the challenges of automated video-documentation of GUI test execution. These concepts were implemented and applied to 370 industrial GUI test cases. The speed and memory consumption was measured in an experiment.

Industry application to real test scenarios at Capgemini has shown that the tailored video-documentation of automated GUI tests is a useful concept for debugging the application under test. It helped in debugging failures. No other known approach offers these capabilities and supports the analysis of GUI tests and the debugging of the applications under test in all aspects discussed above.

Most GUI test frameworks concentrate on the easy description of test cases with specialized script languages or the capture-replay-technique. However, the documentation of the performed operations during test and the task of debugging afterwards still deserve more attention. This work has shown that it can save a lot of time in the testing and debugging phase of a software development process. These aspects are essential for effective and efficient GUI testing. They are widely neglected in existing approaches of video-documentation. The presented concepts provide solutions for the aforementioned difficulties of the task of debugging GUI tests.

REFERENCES

[1] G. J. Myers, The Art of Software Testing, 1st edition. John Wiley & Sons, 1979.

[2] K. Beck, Test Driven Development. By Example. Addison-Wesley Longman, Amsterdam, 2002.

[3] K. Schneider, Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement. Dpunkt, 2007.

[4] "Ranorex." [Online]. Available: http://¬www.ranorex.com

[5] TechSmith, "Camtasia." [Online]. Available: http://-www.techsmith.com/¬camtasia

[6] DebugMode, "Wink." [Online]. Available: http://-www.debugmode.com/¬wink

[7] "Tightvnc." [Online]. Available: http://¬www.tightvnc.com

[8] "Ultravnc." [Online]. Available: http://¬www.uvnc.com

[9] "Ultravnc screen recorder." [Online]. Available: http://-www.uvnc.com/¬screenrecorder

[10] RenderSoft, "Camstudio." [Online]. Available: http://-camstudio.org

[11] Microsoft, "Expression Encoder Pro." [Online]. Available: http://¬www.microsoft.com/¬expression/¬products/-encoderpro_overview.aspx

[12] Microsoft, "Team Foundation Server." [Online]. Available: http://¬www.microsoft.com/¬visualstudio/¬en-us/¬products/¬2010-editions/¬team-foundation-server/¬overview

[13] Microsoft, "IntelliTrace." [Online]. Available: http://-msdn.microsoft.com/¬en-us/¬magazine/¬ee336126.aspx

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," ECOOP'97—Object-Oriented Programming, pp. 220–242, 1997.

[15] T. Sikora, "MPEG digital video-coding standards", Signal Processing Magazine, IEEE, vol. 14, no. 5, pp. 82–100, 1997.

[16] K. Patel, B. Smith, and L. Rowe, "Performance of a software MPEG video decoder," pp. 75–82, 1993.

[17] H. Holzmann, "Videounterstützte Ablaufverfolgung von Tests für Anwendungen mit grafischer Benutzeroberfläche", Bachelor Thesis, Leibniz Universität Hannover, 2011

Article in a journal:

[18] Zimmermann et al., "What makes a Good Bug Report?", IEEE Transactions on Software Engineering, Vol. 35, No. 5, 2010

[19] A. Memon, "GUI Testing: Pitfalls and Process", IEEE Computer, vol. 35, no. 8, pp. 87–88, 2002.

[20] D. Le Gall, "Mpeg: A video compression standard for multimedia applications," Communications of the ACM, vol. 34, no. 4, pp. 46–58, 1991.

[21] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard", Circuits and Systems for Video Technology, IEEE Transactions on, vol. 13, no. 7, pp. 560–576, 2003.

Article in a conference proceedings:

[22] K. Böllert, "On weaving aspects", 1999, 1999 , Proc. ECOOP'99 Workshop on Aspect-oriented Programming, Lisbon, Portugal.