Software Qual J
DOI 10.1007/s11219-013-9206-2

Tailoring video recording to support efficient GUI testing and debugging

Raphael Pham · Helge Holzmann · Kurt Schneider · Christian Brüggemann

Abstract Automated GUI tests typically comprise of several test steps that are executed on the GUI before reaching a point of assertion. Comparing a longer and complex execution of a GUI test to its test instructions for debugging is a laborious task: re-establish the test environment, slow down test execution for human perception, and locate the currently executed test step. Video documentation of GUI tests for debugging purposes is already present in several industry tools. However, it is not optimized for effective documentation of on-screen actions nor synched with the executed test instructions. We present a video-based documentation of automated GUI tests that links the executed test case instruction to the on-screen response of the application under test. Screen recording is optimized for speed and memory consumption while all relevant details are captured. Additional browsing capabilities for easier debugging are introduced. Concepts of aspect-oriented programming are adapted for tracing of pre-compiled test case scripts. 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. The limits of our implementation regarding video capturing and code tracing are explored with a specialized test frame.

Keywords Automated test · Graphical user interface · Video · Code tracing · Aspect-oriented programming

R. Pham (✉) · H. Holzmann · K. Schneider
Software Engineering Group, Leibniz Universität Hannover, Hannover, Germany
e-mail: Raphael.Pham@inf.uni-hannover.de

H. Holzmann
e-mail: helge.holzmann@se.uni-hannover.de

K. Schneider
e-mail: kurt.schneider@inf.uni-hannover.de

C. Brüggemann
Application Lifecycle Service Center, Capgemini Deutschland GmbH, Hannover, Germany
e-mail: christian.brueggemann@capgemini.com

1 Introduction: automated GUI testing

Despite development of different approaches, GUI testing remains a difficult task. Not all technologies support automated calls on GUI elements and often GUI testing is reduced to manual execution of the GUI elements. However, regression testing of test suites has long become an integral part of quality management and maintenance. Also, test-driven development calls for automated regression testing after each change made to the code (Beck 2002). 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. (2010) analyzed what made a good and usable bug report from the perspective of the bug fixing engineer. They found information on how to reproduce the bug to be the most desired. Video recording could provide a usable impression of test case execution. Still the video can barely give insight into 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 Sect. 2, we discuss specific challenges in debugging GUIs. Those challenges are compared to related approaches in Sect. 3. We propose concepts to meet the challenges and present our approach of tailored video documentation in Sect. 4. 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 (Sect. 5). It was then applied to 370 real industrial GUI test cases at Capgemini. In Sect. 4, we consider limitations of our approach. We discuss our findings in Sect. 6 and conclude.

2 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 (Memon 2002). Myers describes this debugging task as two single steps (Myers 1979): 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 (Schneider 2007). To detect a defect, unexpected behavior needs to be analyzed. Often, in GUI tests, 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

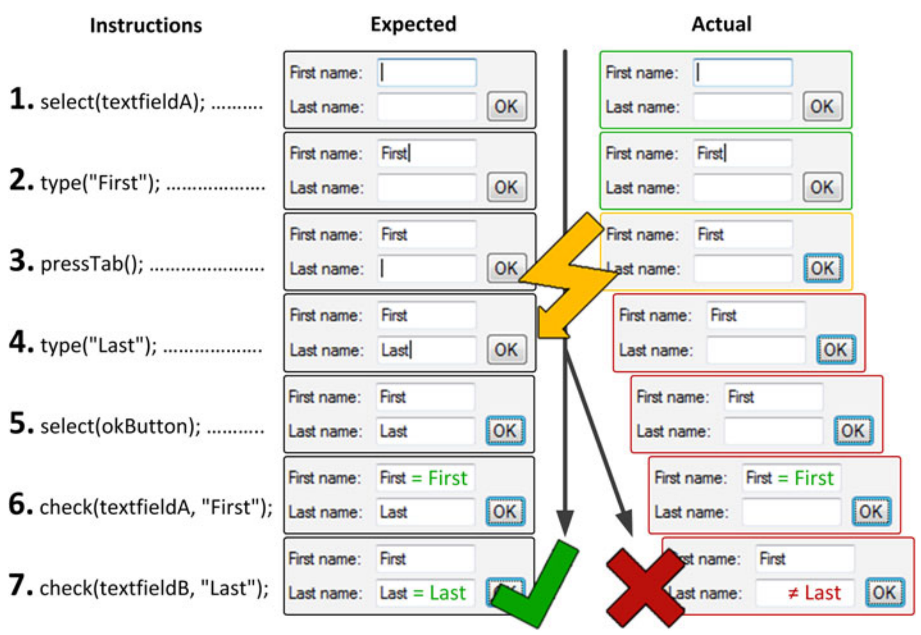


Fig. 1 Deviation of actual behavior from required behavior

from text field A to text field B. The used test framework identifies dialog elements via ID. One GUI test case could be as follows:

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 on-screen. 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 on-screen. 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 on-screen and 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 into the test procedure. In the case of the example, the debugging engineer could quickly recognize test instruction 3 as the cause for the failure—by watching this specialized form of documentation and easily comparing on-screen actions to executed test instructions. Ideally, searching for the failing test step would not require re-execution of the GUI test, and the engineer could start working on the faulty TAB-function right away.

Rather sophisticated mechanisms are required to support this kind of 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. How our approach handles these challenges is evaluated in Sect. 5.3. 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 (2011) do not offer such a scripting language and provide a library to develop the test code in a full-fledged programming language like C#.

3 Related work

We know of no GUI testing tool 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 (2011) 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, and 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 GUIs.

[1] In our semi-structured interviews (Sect. 5.4), GUI testers at Capgemini reported the increase in 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.

Other screen recorders consider the operation of GUI applications by reacting to user inputs to take snapshots. DebugMode's Wink is a working example (2011). 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. The mirror driver works like a usual graphics driver, but without producing any visible output. It can be used to detect the changes on-screen. This concept is often used by virtual network computing software (VNC). The open source project TightVNC software (2011) 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 (2011). The developers of this solution created an own mirror driver, and they also developed a screen recorder using the driver (2011). They have extended the open source CamStudio by RenderSoft (2011).

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 (2011) 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 that allows both of the requested tasks (screen capture and code tracing) is the test framework, which is integrated in the Microsoft Team Foundation Server (2011). It uses the mentioned Expression Encoder to record automated GUI tests and Microsoft's historical debugger IntelliTrace (2011) to simultaneously 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.

4 Concepts of tailored video documentation

4.1 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 (2011)]. 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 (2011) 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 on-screen 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 on-screen change coverage or space consumption, we instigated other event-driven options. To capture a sufficient number 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 mirror driver. 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. Figure 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 that 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 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 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.

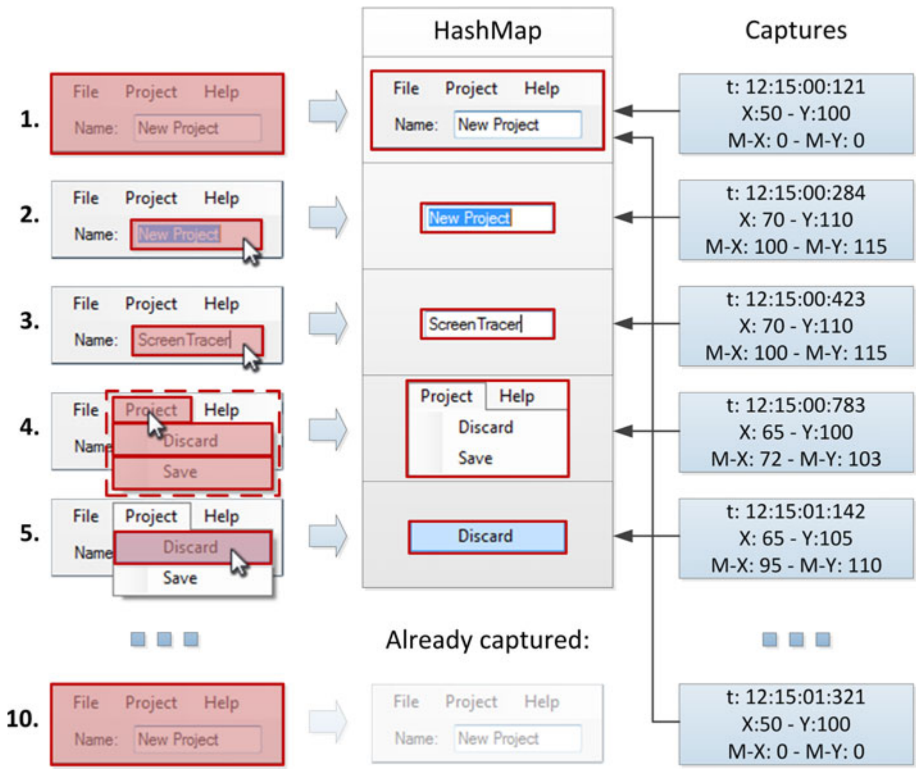
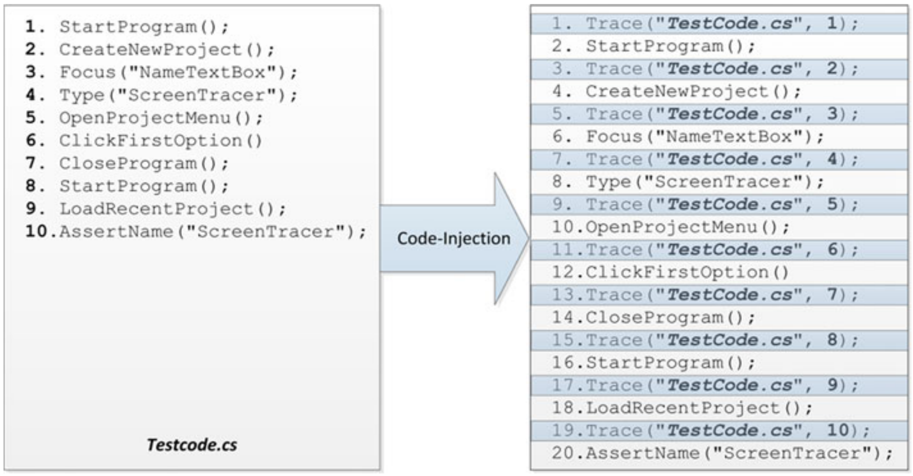


Fig. 2 Output-driven capturing of changed screen segments

4.2 Code tracing

In order to identify and analyze 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. The test code must be traced in order to relate executed test instructions with actions on the screen. This code tracing has to log every test code instruction with the timestamp of its execution. Based on this information, the exact test execution can be reconstructed: Which line of test code has been executed at which point of time? For this purpose, the test code has to be instrumented with specific trace instructions. The instrumentation approach depends on the test framework used as well as on the interpretation policy of test case instructions. Pre-compiled and independent test code is more difficult to trace than code interpreted by a test framework. Generally, code injection based on concepts of aspect-oriented programming (AOP) (Kiczales et al. 1997) can be used to inject code-tracing instructions into the test code. This paradigm separates the code of an application into different concerns. The main functionality of the software is separated from cross-cutting aspects, such as logging or tracing (Laddad 2003; Bodkin et al. 2003). Before the code can be executed, these concerns have to be merged. This process is called weaving. We adapted the techniques of weaving to the above-mentioned code injection. However, typical AOP weaving operates on method level when interweaving concerns. In our case, the cross-cutting concern (logging of execution of a line of test code) called for a more fine-grained approach: line-based weaving rather than only method-based weaving.



```
StartProgram();
CreateNewProject();
Focus("NameTextBox");
Type("ScreenTracer");
OpenProjectMenu();
ClickFirstOption()
CloseProgram();
StartProgram();
LoadRecentProject();
AssertName("ScreenTracer");
```

```
Trace("TestCode.cs", 1);
StartProgram();
Trace("TestCode.cs", 2);
CreateNewProject();
Trace("TestCode.cs", 3);
Focus("NameTextBox");
Trace("TestCode.cs", 4);
Type("ScreenTracer");
Trace("TestCode.cs", 5);
OpenProjectMenu();
Trace("TestCode.cs", 6);
ClickFirstOption()
Trace("TestCode.cs", 7);
CloseProgram();
Trace("TestCode.cs", 8);
StartProgram();
Trace("TestCode.cs", 9);
LoadRecentProject();
Trace("TestCode.cs", 10);
AssertName("ScreenTracer");
```

Fig. 3 Weaving of cross-cutting concern "logging" on line basis

If test code sources are available, trace instructions can be injected easily before compiling or interpreting the code, as illustrated in Fig. 3. This source code weaving is a static weaving approach: Code is merged inseparably during weaving, and tracing can only be disabled by modifying the code again.

In our case, the source of the test code was not available. At Capgemini, the test framework executed pre-compiled C# binaries, and we resorted to so-called dynamic byte-code weaving (Böllert 1999): Code (tracing instructions) is injected into the compiled code, for example, machine code or intermediate language code. In our case, it was essential to keep the byte-code operational despite that late manipulation. For example, the target addresses of branches (which are usually used in assembler or intermediate code) had to be adjusted to work correctly after injecting additional code. Besides the actual weaving, mapping byte-code commands to the lines in the source code proved difficult. Injected instructions needed to trace source code instructions. Respective lines of original source code needed to be linked to the trace. A direct mapping was not possible. Most commands in source code were compiled into multiple instructions in byte code. To resolve this ambiguity, we used so-called symbols. Some compilers can generate symbol files for their corresponding IDE to view the source code on debugging. If no symbol files exist, the source code has to be reconstructed from the compiled byte code by disassembling it.

Capgemini required that the tracing functionality could be enabled or disabled before running a test. We dynamically weaved tracing instructions into the pre-compiled test code at load time. Every time the test frame loads or runs the code to perform the test, tracing instructions are injected dynamically. Its instrumented version exists only temporarily in memory. This was realized by encapsulating the procedures loading the test code and integrating the instrumentation mechanism into them.

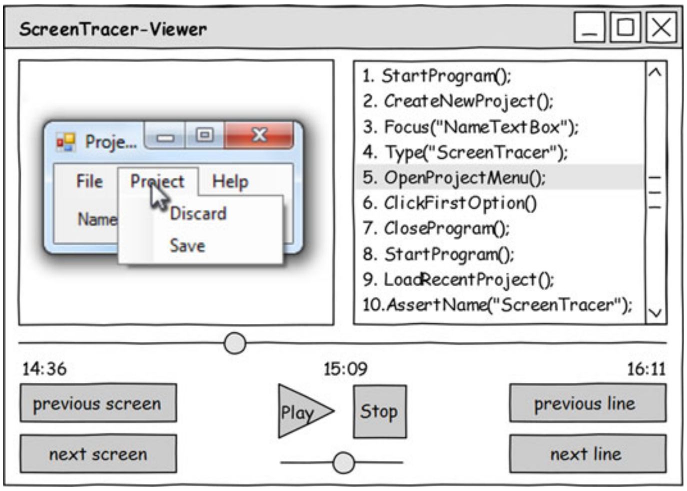
4.3 Viewing

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 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. 4 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 recordings. 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 (Sikora 1997; Patel 1993; Le Gall 1991; Wiegand et al. 2003). 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. 5.



```
1. StartProgram();
2. CreateNewProject();
3. Focus("NameTextBox");
4. Type("ScreenTracer");
5. OpenProjectMenu();
6. ClickFirstOption()
7. CloseProgram();
8. StartProgram();
9. LoadRecentProject();
10.AssertName("ScreenTracer");
```

Fig. 4 Mock-up of the viewer application

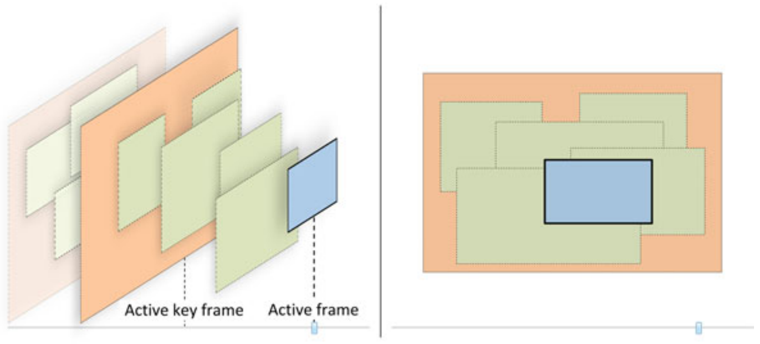


Fig. 5 Screen construction with key frames and segments

5 Evaluation

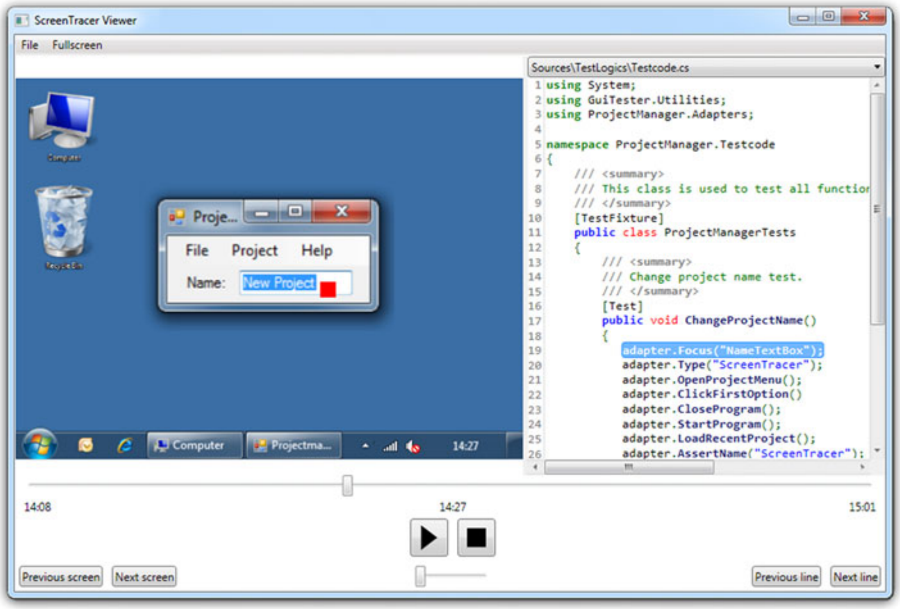
5.1 Evaluation of the operational system

Based on the mentioned deliberations and a performance evaluation (see following Section), an operational video documentation system called ScreenTracer (ST) has been implemented (Holzmann (Holzmann 2011), 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 4 h. The complete test suite of all test cases runs for more than 70 h.

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. 6). The ST and the viewer application were given to debugging developers of Capgemini and are still used 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. In the examined tests, the average size of these segments has been 635 $\times$ 305 pixels, and about 40 % have been duplicates.

Currently, the ST is integrated in Capgemini's test framework and can be enabled or disabled before executing a GUI test that is driven by pre-compiled test code. Our concept of using a mirror driver to capture screen changes and synching these to executed test code instructions can also be implemented in a plug-in for a common IDE, such as Eclipse.[2] However, if the source of the test code is provided in the IDE, instrumentation can be done directly in this source (cf. Sect. 4.2) and will not require byte-code instrumentation.

[2] http://www.eclipse.org.



```
using System;
using GuiTester.Utilities;
using ProjectManager.Adapters;

namespace ProjectManager.Testcode
{
    /// <summary>
    /// This class is used to test all functior
    /// </summary>
    [TestFixture]
    public class ProjectManagerTests
    {
        /// <summary>
        /// Change project name test.
        /// </summary>
        [Test]
        public void ChangeProjectName()
        {
            adapter.Focus("NameTextBox");
            adapter.Type("ScreenTracer");
            adapter.OpenProjectMenu();
            adapter.ClickFirstOption()
            adapter.CloseProgram();
            adapter.StartProgram();
            adapter.LoadRecentProject();
            adapter.AssertName("ScreenTracer");
```

Fig. 6 Screenshot of viewer application

5.2 Comparison of memory consumption

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 on-screen and dismissed them. We tested the user-input-event-driven approach with Wink (2011) and noticed that on-screen 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 test code can contain instructions, which do not result in direct on-screen change, redundant screenshots are taken. Furthermore, actions on-screen, 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 on-screen-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.

Figure 7 shows the memory usage of the time-driven version of the ST in comparison with 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 on-screen. Its maximal frequency of screen captures was 3 frames per second. For

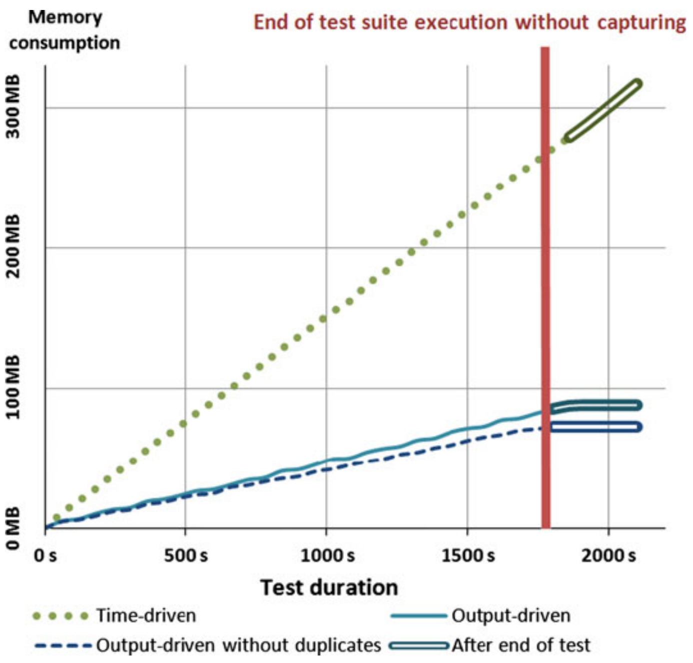


Fig. 7 Memory usage while running a representative test suite of 44 real industry test cases

comparison, the time-driven version took screenshots at a frequency of 3 frames per second. Due to irregular occurrence of changes on-screen, the output-driven version does not capture segments at equidistant time distances (in contrast to the time-driven approach). In order to capture changes on-screen 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 fewer segments and less space.

As shown in Fig. 7, 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, since GUI tests often wait a specified amount of time before proceeding. Slowing a test case down could result in misleading test results.

5.3 Verification of concepts

Development of the ScreenTracer was driven by two goals: tracing the test code and recording changes on-screen. Capgemini intended to use the ST on a large number of existing GUI test cases (written in C#), which would execute automatically and unattended. Regarding challenges for GUI testing (c.f. Sect. 2), we evaluated the limits of the tracing and recording capabilities of our approach. Our research questions were as follows:

1. Completeness of the video recording:

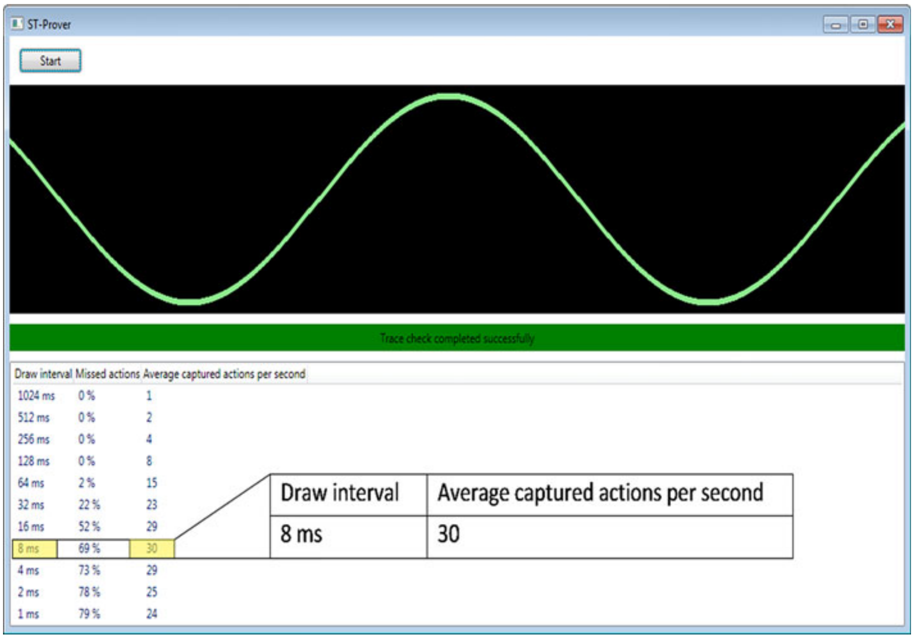
At which frequency of on-screen actions would the ST begin to miss actions? Missing an event on-screen when recording a GUI test could diminish the advantage of the recording, as it is not clear how important the respective action is.

2. Correctness of the tracing information:

Would the ST be able to trace all of Capgemini's real-world test cases? If the ST was not able to fully process and trace the language-specific constructs and instructions, then tracings test runs of several hours could either terminate abruptly or be riddled with untraced parts.

To this end, a measurement application named ST-Prover (Fig. 8) was developed. It acts as a test frame for the ST: It provides the ST with a specialized test code, executes the ST on this code, receives the tracing data and tailored video documentation, and compares these results with expected values. The general test procedure of the ST-Prover is as follows:

1. Use the ST to trace the specialized test code. Obtain tracing information.
2. Simultaneously, use the ST to record the actions on-screen. Obtain tailored video recording of the actions on-screen.
3. Compare tracing information with expected tracing information. Determine how many test code lines were not traced correctly.



Trace check completed successfully

Draw interval	Missed actions	Average captured actions per second
1024 ms	0 %	1
512 ms	0 %	2
256 ms	0 %	4
128 ms	0 %	8
64 ms	2 %	15
32 ms	22 %	23
16 ms	52 %	29
8 ms	69 %	30
4 ms	73 %	29
2 ms	78 %	25
1 ms	79 %	24

Draw interval	Average captured actions per second
8 ms	30

Fig. 8 ST-Prover measurement tool

4. Analyze the video and determine how many generated on-screen changes were recorded or missed.

5.3.1 Completeness of video recording

The general idea is the following: We draw a figure of which we know exactly how it should appear on screen. In this case, the specialized test code draws a sine curve. The curve is drawn pixel by pixel. Each time a new pixel of the sine curve appears, and the ST should register this on-screen change and record it as a new frame in the resulting tailored video documentation. Ideally, the appearance of *each* pixel should be documented by one individual frame of this video. However, if the ST's maximum frequency for registering on-screen changes is exceeded, more than one pixel will appear between two frames of the reconstructed video.

The ST-Prover receives the tailored video from the ST and starts to reconstruct it, capture by capture (described in Sect. 4.3). The ST-Prover determines how many of the generated on-screen actions were captured: Using the mathematical sine function, the ST-Prover computes the coordinates of the spot where the first green pixel (left-hand side of the black drawing area, Fig. 8) should appear. As soon as this pixel appears in the captures of the tailored video, the ST-Prover checks if the next pixel of the sine curve is present as well. If the next pixel of the sine curve has not been drawn yet, this capture recorded the first on-screen action successfully. However, if this screen capture already includes the next pixel (and possibly subsequent pixels) of the sine curve, then this appearance is not the first action on-screen—further pixels have already been drawn and were not recorded. In this manner, the ST-Prover reconstructs the video and analyses if every step of drawing the sine curve has been recorded by the individual screen captures. Figure 9 illustrates this behavior.

If the ST-Prover detects that the recording has missed actions, it traverses the drawn sine curve to its end while counting the number of missed pixels. From there, it proceeds to check the next capture for the calculated appearance of the next pixel. This way, the ST-Prover calculates the amount of recorded actions and missed actions. In Fig. 11 the amount of missed actions between the second and the third capture is two. While the third capture should have included only one new pixel, it also included two additional pixels. Between recording the second capture and the third capture, two of the generated on-screen actions were missed.

However, the black drawing area is not the only area on-screen that changes during the test (e.g. the OS clock is changing as well). By design, these additional changes will be

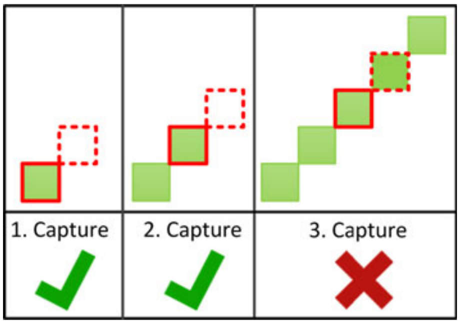


Fig. 9 Pixelwise checking on building up the sine curve. Third capture reveals that three actions on-screen have been missed

recorded and reconstructed in the tailored video as well. In this case, only captures of the black drawing area are relevant and analyzed for verification.

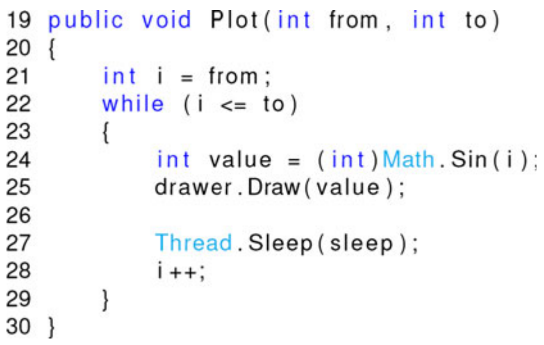
The curve will be drawn several times in varying speeds, in order to find the limit of the screen capturing, e.g. what the maximum speed is before losing on-screen changes. During each run, the coverage of the screen actions is measured and displayed in the lower table of the ST-Prover view (lower part of Fig. 8). Every line represents one drawing speed. Here, “Draw interval” denotes the time between appearances of pixels, in this case the time between on-screen actions. Our findings show that the ST was able to reliably capture up to 8 (fourth row) actions per second on Capgeminis’ workstations. Doubling the number of actions per second results in a negligible missing rate of 2 % (15 captures per seconds, fifth row). In practice, such a low rate is acceptable. The maximal number of recorded actions was 30/s. However, this is accompanied by a very high action-miss rate.

5.3.2 Correctness of the tracing information

The ST-Prover is equipped with the expected tracing information for tracing the specialized test code. This test code is comprised of several methods that use different constructs of the C# programming language. In order to show that the ScreenTracer is capable of tracing all of Capgemini’s test cases, these language constructs were taken directly from these real-world test cases. Each such method draws a short, random part of the sine curve. An example of such a drawing method from the specialized test code is given in Fig. 10. Note, when tracing test code, the ST logs the line number that was executed. In this case, tracing the method in Fig. 10 would traverse line 20, 21, 22, 24, 25, 27, 28, 29 and 22 again (depending on the while-loop). For each such method in the specialized test code, the ST-Prover is equipped with a corresponding checking method (Fig. 11). This checking method verifies the sequence of line numbers that is to be expected as a correct tracing result. The bar in the middle of the main view of the ST-Prover highlights in green (Fig. 8), if comparison between tracing result of the whole test code and expected tracing information is successful. Otherwise, it is red.

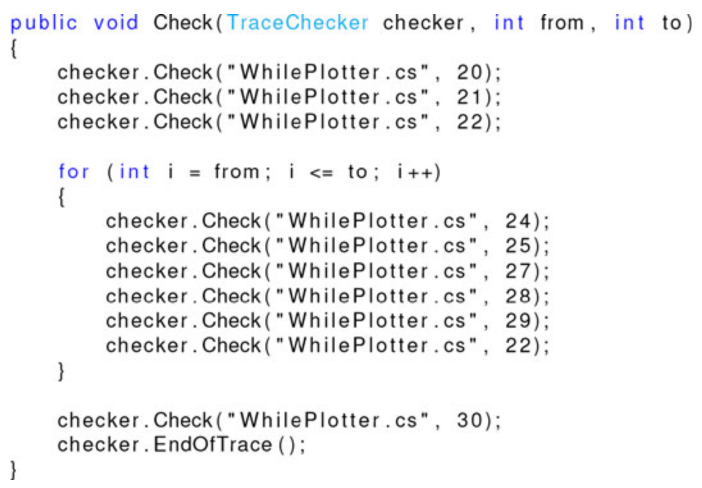
5.4 User reactions

Before the ST was introduced at Capgemini, the test framework in place provided the debugging engineer with a screenshot of the GUI when the failure was detected. In some cases an event log was provided additionally. If the GUI test was done manually, the debugging engineer may have been provided with a textual description of the failure. Capgemini did not use video recording of tests before.



```
19 public void Plot(int from, int to)
20 {
21       int i = from;
22       while (i <= to)
23       {
24             int value = (int)Math.Sin(i);
25             drawer.Draw(value);
26
27             Thread.Sleep(sleep);
28             i++;
29       }
30 }
```

Fig. 10 WhilePlotter.cs, an example of a drawing method using C# language constructs from the specialized test code



```
public void Check(TraceChecker checker, int from, int to)
{
    checker.Check("WhilePlotter.cs", 20);
    checker.Check("WhilePlotter.cs", 21);
    checker.Check("WhilePlotter.cs", 22);

    for (int i = from; i <= to; i++)
    {
        checker.Check("WhilePlotter.cs", 24);
        checker.Check("WhilePlotter.cs", 25);
        checker.Check("WhilePlotter.cs", 27);
        checker.Check("WhilePlotter.cs", 28);
        checker.Check("WhilePlotter.cs", 29);
        checker.Check("WhilePlotter.cs", 22);
    }

    checker.Check("WhilePlotter.cs", 30);
    checker.EndOfTrace();
}
```

Fig. 11 Custom method for checking the correctness of the tracing of the test code method

The ScreenTracer has been in use at Capgemini's for 5 months. We conducted semi-structured, qualitative interviews with two of Capgemini's developers as a first explorative evaluation. These two developers were responsible for debugging GUI tests and used the ST in their daily work process. In interviews of 20 min, we asked them about their experiences of using the ST for debugging real-world test cases. User reactions were generally positive. Both interviewees reported to be able to debug faster and would recommend the tool to other colleagues. We specifically asked about the effects of gaining insights into the test code execution by video documentation. The debugging engineers used the video for rough localization of the beginning of the departure from required behavior. After that, the presented test code is used for fine tuning.

6 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 on-screen 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 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) on-screen 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 on-screen 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 on-screen behavior, which is not directly caused 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.

7 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, common screen recorders as mentioned in related work can capture the actions on the screen during a test and store it in the usual video format. They can be viewed in existing media players with numerous controlling capabilities. However, none of these recorders and media players has 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 on-screen.

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 (2011) of the visual studio, which allow viewing the executed source code after executing it, but none of them provide 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.

8 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 on screen 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. A technical verification of our concepts was performed. It was shown that our approach is able to reliably record up to 15 on-screen actions per second.

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 the test and the task of debugging afterward still deserve more attention. These aspects are essential for effective and efficient GUI testing. They are widely neglected in existing approaches of video documentation. The presented concepts provide a first step toward solutions for the aforementioned difficulties of the task of debugging GUI tests.

References

Beck, K. (2002). *Test driven development. By example*. Amsterdam: Addison-Wesley Longman.

Bodkin, R., Colyer, A., & Hugunin, J. (2003). Applying aop for middleware platform independence. *Practitioner Reports, AOSD*, 2003.

Böllert, K. (1999). On weaving aspects. In Proceedings of the ECOOP'99 workshop on aspect-oriented programming, 1999, Lisbon, Portugal.

DebugMode, Wink [Online]. Available: http://www.debugmode.com/wink. Accessed 2011.

Holzmann, H. (2011). Videounterstützte Ablaufverfolgung von Tests für Anwendungen mit grafischer Benutzeroberfläche. Bachelor Thesis, Leibniz Universität Hannover.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., et al. (1997). Aspect-oriented programming, ECOOP'97—Object-oriented programming, pp. 220–242.

Laddad, R. (2003). Aspect-oriented programming will improve quality. *IEEE Software, 20*(6), 90–91.

Le Gall, D. (1991). Mpeg: A video compression standard for multimedia applications. *Communications of the ACM, 34*(4), 46–58.

Memon, A. (2002). GUI Testing: Pitfalls and Process. *IEEE Computer, 35*(8), 87–88.

Microsoft, Expression Encoder Pro [Online]. Available: http://www.microsoft.com/expression/products/encoderpro_overview.aspx. Accessed 2011.

Microsoft, IntelliTrace [Online]. Available: http://msdn.microsoft.com/en-us/magazine/ee336126.aspx. Accessed 2011.

Microsoft, Team Foundation Server [Online]. Available: http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview. Accessed 2011.

Myers, G. J. (1979). *The art of software testing* (1st ed.). New York: Wiley.

Patel K., Smith B., & Rowe, L. (1993). Performance of a software MPEG video decoder. In Proceedings of the first ACM international conference on Multimedia (pp. 75–82).

Ranorex [Online]. Available: http://www.ranorex.com. Accessed 2011.

RenderSoft, Camstudio [Online]. Available: http://camstudio.org. Accessed 2011.

Schneider, K. (2007). *Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. Heidelberg: Dpunkt.

Sikora, T. (1997). MPEG digital video-coding standards. *IEEE Signal Processing Magazine, 14*(5), 82–100.

TechSmith, Camtasia [Online]. Available: http://www.techsmith.com/camtasia. Accessed 2011.

Tightvnc [Online]. Available: http://www.tightvnc.com. Accessed 2011.

Ultravnc [Online]. Available: http://www.uvnc.com. Accessed 2011.

Ultravnc Screen Recorder [Online]. Available: http://www.uvnc.com/screenrecorder. Accessed 2011.

Wiegand, T., Sullivan, G., Bjontegaard, G., & Luthra, A. (2003). Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology, 13*(7), 560–576.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., & Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering, 35*(5), 618–643.

Author Biographies

Raphael Pham is a research assistant at the software engineering group at Leibniz Universität Hannover in Germany, where he received the Diploma degree in mathematics in 2011. He is currently working toward the PhD degree in computer science under the supervision of Prof. Dr. Kurt Schneider. His research interests include software quality assurance techniques for graphical user interfaces and leveraging phenomena related to testing behavior in open source communities for use in industry.

Helge Holzmann is a master student in the field of computer science at the Leibniz Universität Hannover. From October 2012 to March 2013 he gained experience abroad during a six-month internship at Pivotal Labs in San Francisco, California, USA. Right now he is writing his master's thesis at the L3S Research Center Hannover. Prior to the master's program he wrote his bachelor's thesis about tailored video documentation of automated GUI tests at the software engineering group (SE) of the Leibniz Universität with a practical part performed at Capgemini. This thesis provides the basis for the work presented in this article.

Kurt Schneider is a full professor of software engineering at Leibniz Universität Hannover, Germany. He received his Doctoral degree from the University of Stuttgart, Germany. He held a postdoctoral position at the Center for LifeLong Learning and Design (L3D) at the University of Colorado at Boulder, USA. From 1996 to 2003, he was a researcher and manager at the Daimler research center in Ulm, Germany. His research interests include requirements engineering, software quality and the role of human interaction and communication in software engineering.

Christian Brüggemann studied Computer Science at King's College London and received his MSc for work on Artificial Swarm Intelligence. He spent much time studying graph theory, logic and combinatorial optimization. After his time at King's he joined Capgemini as a Software Engineer, where he worked for two years on large industry projects. Then he decided to become an entrepreneur and founded Graphmasters GmbH, an IT Startup that provides a routing software for navigation systems which actively prevents traffic congestion. Christian is particularly interested in advanced quality assurance techniques such as mutation testing.