Improved Bug Reporting and Reproduction through Non-intrusive GUI Usage Monitoring and Automated Replaying

Steffen Herbold, Jens Grabowski, Stephan Waack
Institute of Computer Science
University of Göttingen, Germany
{herbold,grabowski,waack}@cs.uni-goettingen.de

Uwe Bünting
Mahr GmbH Göttingen
Carl-Mahr-Str. 1, 37073 Göttingen, Germany
uwe.buenting@mahr.de

Abstract

Most software systems are operated using a Graphical User Interface (GUI). Therefore, bugs are often triggered by user interaction with the software's GUI. Hence, accurate and reliable GUI usage information is an important tool for bug fixing, as the reproduction of a bug is the first important step towards fixing it. To support bug reproduction, a generic, easy to integrate, non-intrusive GUI usage monitoring mechanism is introduced in this paper. As supplement for the monitoring, a method for automatically replaying the monitored usage logs is provided. The feasibility of both is demonstrated through proof-of-concept implementations. A case-study shows that the monitoring mechanism can be integrated into large-scale software products without significant effort and that the logs are replayable. Additionally, a usage-based end-to-end GUI testing approach is outlined, in which the monitoring and replaying play major roles.

1. Introduction

Most of today's software applications are driven by the interaction between the users and the Graphical User Interface (GUI) of the software. Many bugs are triggered as the result of a chain of interactions between the user and the GUI. An important part of resolving bugs is to replicate them and understand by what they are caused. Therefore, detailed information about the interaction with the GUI is an important means of the bug fixing process [1]. However, in most cases, the person who discovers a bug and the one fixing it are not the same. When customers experience a bug, they hopefully report it to the developers. The first task of the developer is to reproduce the bug in order to confirm it. The difficulty of this task depends to a large degree on the available information about the customers interactions with the application that triggered the bug. However, the manual listing of the exact interactions with the application is a tiresome and error-prone procedure. Therefore, a customers bug report usually lacks information about detailed user input. One way to resolve this are automatically generated reports that include the detailed user inputs and that can be attached to bug reports.

In this paper, a generic usage monitoring mechanism is introduced that does not require significant change of source code, regardless of the size of the application that shall be monitored. Thus, it can be integrated into already existing applications at low costs and with little manual effort. The mechanism itself resembles the capture phase of *capture/replay* approaches for GUI testing [6] in principle. However, there are several key features of the monitoring approach presented in this paper that make it unique. The presented mechanism can be integrated directly into the software and monitor the usage *from the inside*, in contrast to recorder tools used for capture/replay that are seperate applications. It can only monitor the applications it has been integrated into in comparison to recorder tools that can potentially monitor any application in the system. While this seems like a drawback, it actually allows the deployment of the monitoring to customers without raising too many security concerns.

Additionally, no assumption about optional properties of GUI objects are made. For example, the Java[1] tool Abbot/-Costello for testing Java GUIs assumes that each of the GUI

[1] http://www.oracle.com/java

components has a symbolic name. Another key feature of our approach is that not only user interactions like mouse clicks and keyboard input are monitored but also important internal commands, i.e. part of the softwares internal communication. This further enriches the information for developers who look for possible bugs, as it allows them to analyze whether commands are sent as expected.

By integrating this usage monitoring mechanism into deployed software, *continuous monitoring* is possible. In this case, there always exists a detailed log of the user interactions when a bug occurs. Customers only have to supply the log instead of manually explaining what exactly they did. To manage the size of the logs circular buffering is used, i.e., information is only stored for a fixed amount of time and is removed afterwards. The amount of time may vary depending on the software under surveillance. The drawback of using circular buffering is that the cause for failures might be to far in the past and is already removed.

To further support for maintainance process, an automated method for the exploitation of the monitored usage data is presented in this paper. If a bug is reproduced and verified, it needs to be specified how exactly it can be reproduced. In the best case, developers create an automated script that triggers the bug. This script can then be used as a test case to evaluate if the bug has been fixed. To further support bug reproduction, an automated replay mechanism is defined. In general, this replay is similar to that of existing capture/replay techniques. The aim is not only to accuratly replay the monitored usage logs, but also to be independent of absolute screen coordinates. While this is a rather simple task for some GUI interactions, for others it can be difficult as they rely heavily on screen coordinates, i.e., the location of the mouse pointer and GUI objects on the screen.

On the one hand, to replay a button click, only the button needs to be located, afterwards the click can be performed. On the other hand, consider dragging a scrollbar with the mouse. Only locating the scrollbar and sending messages that it has been clicked is not sufficient, as the mouse movement during the clicking needs to be accounted for. However, the mouse movement depends on screen coordinates. In this paper, the abstraction from mouse movement using internal messages is presented. This is possible as the mouse movement does not move the scrollbar directly, but it rather triggers the generation of internal messages of the program that cause the movement. It is shown, how these indirectly generated messages can be used to replay usage of the application without relying on screen coordinates.

Both mechanism have been prototypically implemented and used in a case study to demonstrate their feasibility.

The contributions of these paper are twofold: 1) a generic, easy to integrate, non-intrusive method for monitoring user actions; 2) a novel replay mechanism for previously monitored user actions.

The structure of this paper is as follows. The monitoring and the replay mechanism are introduced in Sections 2 and 3. In the following, a proof-of-concept implementation is introduced in Section 4. Afterwards, Section 5 introduces a case study performed to validate the feasibility of the monitoring and replaying mechanisms. In Section 6 a usage-based end-to-end GUI testing framework is outlined that will be the defined and implemented as future work. In Section 7 the related work is discussed. Finally, in Section 8 the paper is concluded and an outlook on future work is given.

2. Passive GUI Usage Monitoring

A primary objective of this work is the design of a generic and easy-to-integrate GUI usage monitoring mechanism that is non-intrusive and can thus be deployed to customers. To this aim, the mechanism needs to be able to log all user interactions with the GUI. The mechanism developed as part of this work is aimed at Microsoft Windows programs using the Microsoft Foundation Classes (MFC)[2] as target platform. We believe the mechanism is also adaptable to other platforms, such as X-server based GUI-kits, as they provide similar mechanisms.

Before the monitoring concept itself can be introduced, two key features of the Windows Application Programming Interface (API) must be explained: *window handles* and *messages*. Each GUI object in Windows, be it an edit box, a button, or the mainframe of a window has a unique window handle, referred to as *HWND*. The HWND is not only unique for the GUI process, but system-global, i.e., no two GUI object in the whole system have the same HWND. When a GUI object is created it is assigned a free HWND. After its destruction the HWND may be reused. Therefore, the "same" GUI object will very likely have different HWNDs in different program executions.

Messages are used for the communication with GUI objects. This includes communication of the operating system with GUIs, communication between the programming logic and the associated GUI as well as communication between GUI objects. The messages have a *type* and two parameters. How the parameters are used depends on the message type. To link the messages to a specific GUI object, HWNDs are used. Examples for messages are `WM_CREATE`, which is sent when an GUI object is created or `WM_LBUTTONDOWN`, which is sent when the left mouse button is pressed down. The messages that are relevant for this work can be roughly divided into three categories: 1) creation, destruction, or alteration of GUI objects; 2) user interaction with the GUI; 3) internal commands.

[2]http://msdn.microsoft.com/en-us/library/d06h2x6e.aspx

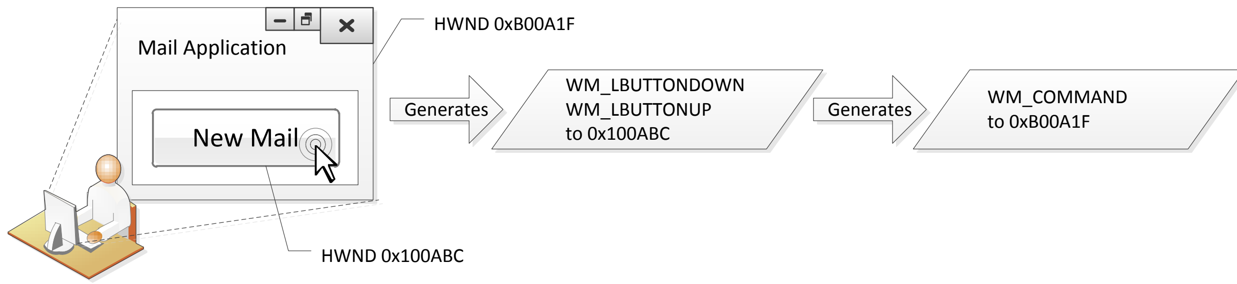

Figure 1. Messages generated by a button click

The first category enriches the log with information about the current state of the GUI, e.g., which GUI objects exists, their parents, or whether they are modal[3]. Developers can use these informations to analyze if windows have been created and destroyed properly. Examples for such messages are WM_CREATE, WM_DESTROY, and WM_SETTEXT, which are sent when a GUI object is created, destroyed or its title is changed respectively.

The second category consists of messages generated by the operating system as a result of mouse or keyboard activity by the user. This includes mouse movement, mouse clicks, and keyboard input. Examples of such messages are WM_LBUTTONDOWN for "left mouse button down", or WM_KEYDOWN when a key is pressed. The latter uses parameters to indicate which key has been pressed. For most software, the mouse movement itself has no effect and it is rather driven by mouse clicks and keyboard input only. Therefore, mouse movement is often irrelevant for the usage monitoring. Thus, the logging of mouse movement is optional, as it generates lots of messages clogging the log.

Monitoring messages of the third category is the strength of the mechanism. The internally sent messages are used for communication among GUI objects. For example, a button telling its parent window that it has been clicked. Internal commands are, e.g., WM_COMMAND messages. These messages are used by the application's command handlers to control its activity. By logging internal commands, it is not only possible to record how the user interacted with the GUI, but also which actions were triggered internally as a result. This allows developers to check if messages are sent as expected or if messages are adressed wrongly or missing. Monitoring of this kind of message is a crucial feature exploited by the replay mechanism outlined in the following section.

In Figure 1, a simple example of the message generation and sending process is shown. Consider a click on a *New Mail* button. The operating system then generates two messages, WM_LBUTTONDOWN and WM_LBUTTONUP and sends them to the button using its HWND. As a reaction to these messages, the button generates a WM_COMMAND message to its parent, i.e., the *Mail Application*. The application then handles the command accordingly, i.e., initiates the procedure to write a new mail. This example shows how internal messages are generated indirectly as a result of user actions. Current GUI usage monitoring tools only capture the WM_LBUTTONDOWN and WM_LBUTTONUP messages, the novelity of our approach is to also capture the generated WM_COMMAND message.

The monitoring concept itself is based on *message hooks*[4]. A hook can be placed in the message processing queue of a process to intercept messages. Each process has one message processing queue, that is used by all GUI objects of the process. Therefore, hooks provide an elegant method to monitor the internal communication of a process in a centralized way, without altering the rest of the source code. The hook procedure has two tasks.

The first is to determine which messages are logged. Simply logging all messages is infeasible as windows application send thousands of messages internally, most of which are irrelevant for the purpose of usage monitoring. For example, most widgets are constantly polled for their name with the WM_GETTEXT message. This rapidly amounts to thousands of such messages. However, these messages are irrelevant for the monitoring. If all of these messages were to be logged, it would produce huge amounts of data and the applications performance would drop dramatically. A small experiment where all messages were logged and in which only a small test application was started an two clicks performed already produced 13 MB of log data, and the application was slowed down to a degree where using it was infeasible. Therefore, the hook procedure must implement a message filter and record only the three categories of messages described above. This filter is a simple pass-through filter based solely on the message type: messages of the three categories described above, like WM_LBUTTONDOWN and WM_COMMAND, pass through; all other messages, like WM_GETTEXT, are filtered from the logging. With this filter in place, the same experiment only produced 10 KB of

[3] A dialog blocking the rest of the application. Detailed explanation at http://msdn.microsoft.com/en-us/library/aa984358.aspx

[4] http://msdn.microsoft.com/en-us/library/ms632589.aspx

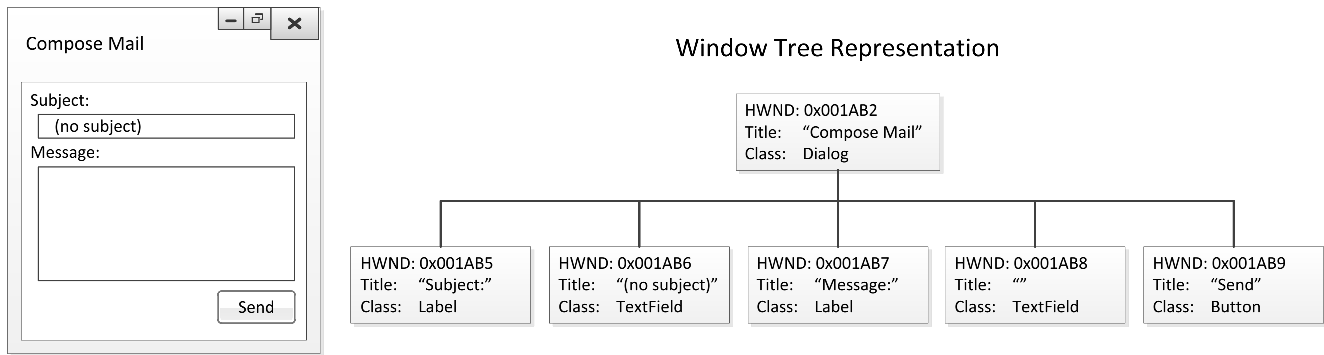

Figure 2. A dialog and its window tree representation

log data and the application performed normally.

The second task of the hook procedure is not only to log the messages themselves, but include information essential to them. This can be additional information about the internal state of the program or the GUI object that a message has been sent to. For example, when a GUI object is created, its title is not included in the `WM_CREATE` message. Instead, the title must be obtained by querying the created object.

3. Replaying User Actions

In the following, a replay mechanism based on logs generated by the previously defined usage monitoring mechanism is introduced. The replay is robust to GUI layout changes, resizing operations, and different screen resolutions, because it does not depend on screen coordinates. This improves the replayability of logs recorded on different work stations. It will be defined how operations that usually depend upon screen coordinates can be performed in an indirect way without relying on the coordinates. The mechanism can be split into two main components: a representation of the software's GUI state extracted from the log and the replaying of actions itself.

The internal representation of the GUI state is used as a means to match the GUI objects observed during the monitoring to the GUI objects during the applications replay. The state is represented in form of a window tree, where the GUI objects are represented as nodes. The parent/child releationships between the GUI objects are represented by the tree's structure: top-level windows are children of a virtual root node; all other GUI objects are children of their respective parent. As each GUI object has a unique HWND, the HWND is used as an ID for the nodes in the tree. In Figure 2 an example of a window tree for a dialog is given. The elements of the dialog are its children in the window tree. For the creation and maintainance of the window tree the messages of the first category are used, i.e., those related to the creation, alteration, and destruction of widgets. Each time such a message is found in the log, the window tree is adapted according to the message and thus remains consistent. For example, if a `WM_CREATE` message is found, the created GUI object is added to the tree. The nodes of the tree contain information about the widget they represent, like the name and the class of the GUI object. If information about a GUI object in the log is required, it can be looked up in the window tree using its HWND. The window tree provides a means to have a *dynamic* representation of the GUI, as it always describes the state of the GUI "as is" at all times.

The principle of replaying is rather simple: execute every recorded user action exactly once and in the order they have been recorded. As simple as this sounds, especially the "exactly once" part is problematic. Consider "exactly" as "at least once" and "at most once" at the same time. To achieve "at least once", for every action a replayable message (or sequence of messages) that triggers the action must be determined. If simply all messages are replayed, this citerium is certainly fulfilled. However, if all messages were to be replayed, some actions would almost certainly be triggered more than once. The reason for this are two characteristics of the messages. First, there are many internal messages that are automatically generated. To replay these messages may result in unwanted behavior as the message would be duplicated. Second, a message generated as a result of a user action almost certainly triggers further actions, as it is depicted in Figure 1. If both the user action and the triggered action were to be replayed, the result of the action would be executed twice. In the example, the `WM_COMMAND` message would be sent twice. Therefore, the "at most once" part would be violated. Another remedy would be to restrict the replay only to the messages directly resulting from user actions, i.e., `WM_LBUTTONDOWN`, `WM_KEYDOWN` and so on. As these messages often depend on screen coordinates, this would violate one of the goals of the replay mechanism. Thus, a

sophisticated method to select which messages must be sent is necessary to meet the requirements.

To meet the principle of "exactly once", it is translated to "one action triggered per user interaction message". There are two general cases: either the user interaction can be directly replayed without relying on coordinates or not. In the first case that is exactly what the replay does. The second case is rather complex. If a non-replayable message is found, the replay switches into the state "an action must be replayed, but it is not known which yet", including where it should have been executed. No action is executed yet. Instead, the replay just continues evaluating the logged messages and looks for *actionable* messages, e.g., command messages. Such actions are normally triggered directly as a result of the user input. Therefore, to *perform a user action without the user interaction*, the actionable message is replayed instead of the interaction. The prime examples for actionable messages are `WM_COMMAND` messages. In case a `WM_COMMAND` is found, it can be checked if the HWND of the command's source is the same as the HWND where the user action occured. If this is the case, the appropriate actionable message is found and it is sent to emulate the user action.

4. Proof-of-Concept Implementation

The mechanism defined for usage monitoring and replaying of message logs defined in sections 2 and 3 are implemented in two prototypes. In the following, these prototypes are introduced and important implementation details are discussed.

4.1. Monitoring

The monitoring prototype was designed to be easily integratable into any Windows application. To reach this goal, it has been implemented as an independently loadable component in a Dynamic Link Library (DLL)[6]. The loading of a DLL is a simple task with little manual effort. Thus, a DLL fits the requirement "easy-to-integrate". The interface of the DLL is kept simple and consists only of functions to enable/disable the monitoring, which further simplifies the integration of the monitoring.

The monitoring writes the logged information as textual XML snippets. For each logged message a `msg`-node is added to the end of the log. The type of the message is stored as an attribute of the node. Each message can have an arbitrary number of `param` nodes as children to store additional parameters important for the message. The parameters themselves have two string attributes, containing the type of the parameter and its value. In Figure 3, an exemplary log excerpt is depicted. It shows the results of a click with the left mouse button. After the messages `WM_LBUTTONDOWN` and `WM_LBUTTONUP` for the mouse click itself, a command message is generated as result of the click. As it can be seen in the `source` parameter of the command message, the source of the command is the same as the HWND of the GUI object that was clicked. This command triggers the creation of a new window. There are four parameters recorded with the message that describe the properties of the created window: its HWND, its parents HWND, its resource ID, and its class. These parameters are not the same as the parameters of the `WM_CREATE` message during the program execution. For example, the `window.parent.hwnd` parameter is determined using the `GetParent()` function of the Windows API. The logs themselves are stored as a time-depending circular buffer, i.e., after a user-defined amount of time, the monitored data is destroyed.

To wiretap the internal message communication, hooks are used to intercept messages. The proof-of-concept implementation uses two hooks: `WH_GETMESSAGE` and `WH_CALLWNDPROC`. The usage of both hooks is mandatory, as both hooks do not intercept the whole internal communication related to user interaction, only parts of it. However, the messages recorded by these hooks are not disjunctive. Therefore, some messages are either recorded redundantly or a message filter needs to be employed. As the latter would reduce the perfomance of the monitoring, the messages are recorded redundantly to keep the monitoring lightweight. Instead, the replay needs to be able to identify and ignore redundant messages.

The two hooks are *process hooks*, i.e., they can only intercept messages that are sent to the process that created the hook. Therefore, it is not possible to accidentally monitor other applications. This is in contrast to using *system-global hooks* that can intercept all messages. While it might seem prudent to use global hooks for usage monitoring, as to "not miss anything", there are two important reasons for using process hooks. The first is that global hooks decrease the performance of the whole system, as each and every message will be processed by it. Secondly, global hooks are less likely to be accepted by customers. They can potentially monitor the whole system (including all keystrokes!), which might be looked upon as a security risk. Of course, the monitoring of only the target application itself also raises security and privacy concerns. However, the implementation provides functions for enabling and disabling the monitoring at any time. Thus, a configuration dialog can provide users a simple means to disable the logging to protect their privacy or to not raise security concerns and only en-

[5]The message types, like `WM_CREATE` are actually only C++ preprocessor defintions, for example, `#define WM_CREATE 1`

[6]http://support.microsoft.com/kb/815065/EN-US

```
<msg type="513">
   <param name="window.hwnd" value="919280"/>
</msg>
<msg type="514">
   <param name="window.hwnd" value="919280"/>
</msg>
<msg type="273">
   <param name="window.hwnd" value="330554"/><param name="command" value="31067"/>
   <param name="source" value="919280"/>
</msg>
<msg type="1">
   <param name="window.hwnd" value="657830"/><param name="window.parent.hwnd" value="789102"/>
   <param name="window.resourceId" value="1034"><param name="window.class" value="#32770"/>
</msg>
```


Figure 3. A listing of messages generated by a mouse click[5]

abling it if they encounter problems with the application. Furthermore, the circular buffering can be used to prevent long-term data about users, e.g., by setting it to one day.

4.2. Replay

The replaying prototype is implemented as a stand-alone command line application. To process message logs, event-based SAX[7] parsing is used. The messages are handled one at a time without lookahead. Messages of the types `WM_CREATE`, `WM_DESTROY`, and `WM_SETTEXT` are used to maintain the window tree. The tree node of each GUI object stores its HWND, name, class, resource ID, and whether it is a modal dialog.

When a user interaction is found in the log, the tool decides if it can be replayed directly or if it must be replayed indirectly using actionable messages. This decision depends on the type of user interaction and the GUI object it has been performed on. When this heuristic decides to send a message – be it a user interaction or an actionable message – the `PostMessage()` function of the Windows API is used. It provides means to send messages to any GUI object in the system, given its HWND. As the function sends messages asynchronously, the replay tool pauses after sending a message for 500 miliseconds to prevent flooding the application that is target of the replay with (premature) messages. Timing the delays more accurate, is not supported by the prototype.

To obtain the HWND of the target GUI object, the window tree is used. This is done by matching the ancestry of the target GUI object to the windows currently existing in the system. To this aim, the `EnumWindows()` and `EnumChildWindows()` procedures of the Windows API are used. The `EnumWindows()` procedure enumerates all top-level GUI objects, the `EnumChildWindows()` procedure enumerates all child objects of a given GUI object. To match the windows, the ancestry is traversed from parent to child and at each level a score is assigned to the enumerated windows. The score depends on the GUI objects resource ID, name, and class. The scores are depicted in Table 1. On the top-level and for modal windows, all currently existing top-level windows are enumerated. Otherwise only the child objects of the previously found GUI objects with the highest score are considered. If the score is lower than 3 a warning is logged, as the similarity is rather small resulting in possible errors, e.g., if only the name "Ok" is matched. If the score is 0, there is no existing GUI object that matches target GUI object and the replay aborts with an error.

Equal attributes	Score
resource id, name, class	6
resource id, name	5
resource id	4
name, class	3
name	2
class	1
–	0

Table 1. Scoring function

The proof-of-concept implementation of the replay mechanism only supports a subset of user interactions with the system. This includes actions performed on buttons, radio and check boxes, edit boxes, slider, tool bars, scroll bars with both mouse and keyboard input. Furthermore, system commands based in keyboard input are supported, e.g., the popular "Alt+F4" key combination used to close applications. Interaction with menus is only partially supported.

[7] http://www.saxproject.org

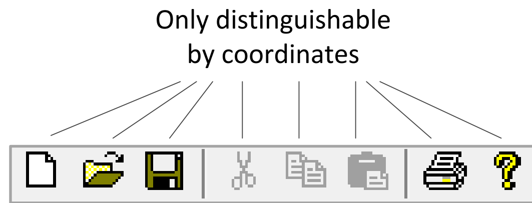

Figure 4. Toolbar

It is possible to trigger actions from menus, e.g., open an about box by clicking on “Help” → “About”. However, the replay does not actually open the menu, instead only the action, i.e. open the about dialog, is performed.

The case of clicking on tool bars exemplifies the strength of using internal messages. Normally, the tool bar resolves which icon has been clicked using the screen coordinates of the mouse during the click and thereby determines the action. The tool bar icons themselves have no identifiers like the HWND (see Figure 4). Thus, replaying tool bar actions normally requires screen coordinates. However, clicking on a toolbar triggers an internal message, e.g., `WM_COMMAND` that can be observed and used to replay the action without reliance on screen coordinates.

5. Case Study

As part of a cooperation with Mahr GmbH Göttingen (Germany), the usage monitoring and replay prototypes have been tested with the software *MarWin*. MarWin is a large scale industrial software platform, designed to be the basis of software products for both existing and future measuring devices in the field of dimensional metrology. It is written in C++ and consists of more than 2 million lines of code. Several hundreds of products based on MarWin are shipped each year. The software is subject to continuous development and long-term evolution.

The foremost goal was the integration of the usage monitoring prototype into the software. The first step for the integration was to load the DLL that implements the monitoring and to enable logging at the start of the application and afterwards to disable the logging and to unload the DLL at its termination. Including error checking, only 15 lines of code needed to be added to achieve this. Afterwards, the usage monitoring is fully functional. The second part of the integration into MarWin was to utilize an already existing tracing mechanism. The log file already in place was to contain the additional information. The resulting log is a mixture of trace messages and usage information. To differentiate between the two, the prefix “UL:“ is used to identify usage information. For this purpose, a patched version of the monitoring DLL was created. As only the actual writing of the information needed to be modified, the changes were also minimal with 12 lines of code. Altogether, only 27 lines of code needed to be modified to fully integrate the usage monitoring into an industrial software.

To determine the impact of the monitoring on the performance, we modified version of the monitoring DLL, to count the number of messages received (before filtering), the number of messages logged (after filtering) and to measure the time consumed by the monitoring. In an experiment, 64 user actions were performed. A total of 20.221 messages were received of which 506 were recorded. The time consumed for the monitoring was 1361 milliseconds. In average, the time consumed to record a message is about 2.7 milliseconds and 21 milliseconds per action. The size of the generated log was 139 Kilobytes. These numbers show the importance of the message filtering, as only 2.5% of the messages are actually recorded. Furthermore, the average time per action is with 21 miliseconds very low. Thus, the monitoring does not impact the performance negatively.

The second goal was to validate the replaying mechanism with large-scale industrial software. In many aspects, such software is different from small-scale projects laboratory projects. With regard to GUI testing, there are many more widgets, which makes locating them for the replay more difficult. Furthermore, the widgets are costumized to a higher degree and thus deviate more from standard examples. Even so, the proof-of-concept implementation of the replay can be used successfuly to execute the currently implemented actions with Mahr’s software.

6. Future Work

In the long run, the usage monitoring and the replaying of the monitored logs shall be part of a usage-based approach towards an end-to-end GUI testing framework. There are two big challanges associated with GUI testing. The first is insufficient automation of the testing process [8]. The second problem is that GUI-based applications often have a very large state space. Exhaustive testing is time and cost expensive and therefore infeasible. Even more so due to the poor automation. Therefore, GUI testing is often done in an ad-hoc and manual fashion. To cope with these issues a usage-based framework is suggested. Usage-based testing has successfully been applied to other event driven software, like Web applications [4, 12, 14] and to a lesser degree also GUI testing [2]. The testing framework is based upon the integration of three mechanism and the resulting synergy effects: 1) GUI usage monitoring; 2) replaying of usage logs; 3) stochastical usage models. The first two have been introduced in this paper, the third is outlined in the following.

Stochastical usage models are a means to describe user behavior. The models can, e.g., be based on Markov-Models (MMs). Figure 5 shows an examplary usage model

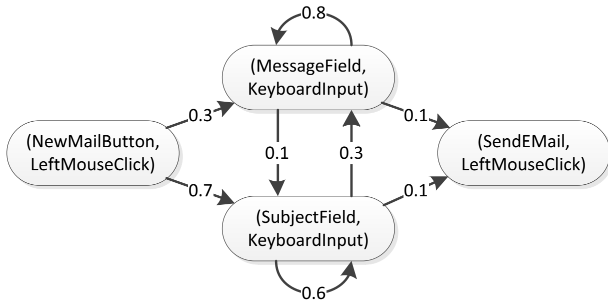

Figure 5. Examplary GUI Usage Model

of a simple e-mail application. Based on this example, the features and capabilities of such models will now be explained. The example is a first-order MM, i.e., in every state of the model, there is a fixed probability for each possible state transition that does not depend on the previous actions. In other words, in such a model, the users do not remember their last actions, only where they currently are. The states of the model are tuples describing the last user action: the action that was performed and the GUI object it was performed on. For example, a click with the left mouse button on the new mail button. The labels on the state transitions define the probability with which action will take place next. For example, after keyboard input in the subject field, there is a 60% chance that there will be further input, a 30% chance that there will be keyboard input in the message field, and a 10% chance that the user will click on the send button. These transition probabilities are estimated by analyzing usage data. The states of the model are either modeled manually by an expert or extracted from usage data. While a first-order MM is used in this example, more sophisticated models will be used in our work, e.g., n-th order MM, Prediction by Partial Match (PPM) [3], or Context-Tree Weighting (CTW) [15].

For end-to-end testing several tasks need to be performed. To improve the automation, the framework tries to tackle the difficulties of GUI testing as a whole, not only small portions seperatly. For example, existing capture/replay tools are capable of automating the GUI test execuction. Otherwise, they are poorly integrated into the testing process, e.g., with test case generation. Vice versa, a test case generation approach may generate execution paths to be tested, but not scripts to be used by replay tools.

In Figure 6, the envisioned framework is outlined. The highlighted part is covered by this paper. Through *Monitoring of Usage*, *Actual Usage Traces* are gained that can be added to *Bug Reports* and used as input for the *Replaying of GUI Actions*. Furthermore, *Stochastical Usage Models* can be trained based on the actual usage traces. The stochastical model is then used to *Randomly Generate Traces*. This a test case generation mechanism and the traces can be executed using the the replaying mechanism. By using the monitoring mechanism during the replay, a *Traces of Replay* of the test cases are generated. Through *Comparison* between the replay traces and the input traces, replay test verdicts (e.g., pass and fail) can be determined. If the usage information is insufficient to generate verdicts, tracing information can be added to the log – as has been done in the case study – to allow further comparison. Additionally, *Usage Statistics* about the software can be gained by analyzing the actual usage traces and stochastical usage models.

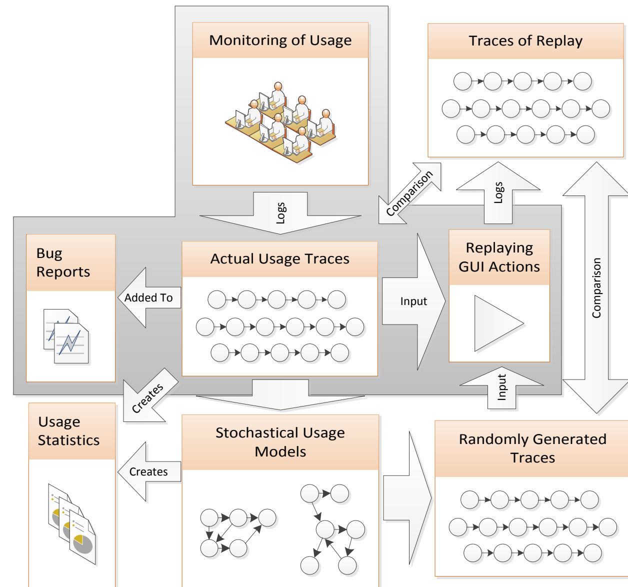

Figure 6. A usage-based end-to-end GUI testing approach

As the outline shows, by using the three mechanism for usage monitoring, replaying and stochastical modeling, many testing tasks can be automated.

7. Related Work

The usage monitoring approach and replaying mechanism presented in this work have many similarities with existing capture/replay techniques.

The capture/replay tool jRapture is based on the a modified version of the Java API [13]. By replacing parts of the Java API with a modified version, all kinds of input are monitored, including mouse events, keyboard events and file input. This mechanism is similar to the hooks used in this work. However, in contrast to this work, a virtual machine and bytecode are manipulated and the monitoring is not directly integrated into the application. Furthermore, it is vulnerable to changes to the Java API, e.g., if a new class is added, the modified API would not monitor its usage.

This work proposes the use of elemental API functions and is thus more stable towards API changes.

Another approach similar to hooks is presented as part of the GUITAR[8] suite [9]. The authors use Java's reflection mechanism to manipulate event and action handlers used by Java GUI objects, by wrapping own handlers around the already existing ones. The new handler then performs the monitoring and delegates the actual handling of the action to the original handler. This is the same concept as used in this work based on hooks: the events – in this case messages – are intercepted to perform the monitoring but are otherwise handled as usual. In contrast to the monitoring presented in this work, the monitoring of internal events and extraction of information about the GUI structure are not parts of their work.

A different approach to define a deployable monitoring approach for capture/replay applications is to use instrumentation to modify the deployed code [10]. Instead of monitoring the user interaction, method calls and field manipulations are logged. The generated binary code is changed through instrumentation, by modifying method signatures and field manipulations. In comparison to our approach, this is highly intrusive as it changes the actual binary code of the application instead of deploying the monitoring as a seperate component. Additionally, the monitoring is on a lower level of abstraction, as all method calls are monitored and not only those related to the software's usage.

The *HP WinRunner* is a capture/replay tool and part of a functional software testing suite [5]. It uses a *GUI Map* that is similar to our window tree. However, the GUI Map is obtained by inspecting the GUI objects as they are opened and is otherwise static. This is a difference to the window tree, that is dynamic and changes as the GUI changes during the recorded usage. Another structure similar to the window tree is the *GUI forest* [7]. The GUI forest is a structure of all windows of the application and can be obtained by using a tool called GUI Ripper. Same as the GUI Map, the GUI forest is static and does not reflect the current state of the application, but rather what objects could possibly exist in the GUI.

A different approach towards GUI testing was developed by [11]. They defined a model-based GUI testing methodology for the .Net platform as an extension of the Spec Explorer[9] tool for model-based specification and testing. To this aim, they describe the GUI as a Finite State Machine (FSM) from which test cases can be derived automatically with Spec Explorer. Furthermore, they provide a GUI mapping tool to support the task of mapping the FSM to the actual implementation of the GUI.

[8]http://guitar.sourceforge.net

[9]http://research.microsoft.com/en-us/projects/specexplorer

8. Conclussion and Future Work

In this paper, a generic, non-intrusive, easy-to-integrate usage monitoring mechanism for message-based GUI systems was introduced. Based on the usage logs produced by the monitoring, an automated replay mechanism was defined and implemented. The feasibility of both the monitoring and the replay mechanism has been demonstrated by means of proof-of-concept implementations. In a case study, the monitoring prototype was successfully integrated into a large-scale industrial software with little effort. In this industrial setting, the replay was successfully applied. Furthermore, a usage-based end-to-end GUI testing approach was outlined and the role of the monitoring and the replaying in this approach was described.

Future work on this project is manyfold. On the one hand, the replaying mechanism will be extended to be able to replay a larger set of actions, with the aim of being able to fully replay all logs. To reach this goal, the monitoring will also be extended to enrich the log with further data about the internal communication. In addition to the work on replay and monitoring, the stochastical usage models will be defined and implemented, including a method for the estimation of the model parameters from usage logs. Furthermore, the outlined GUI testing approach will be defined in detail, e.g., how the stochastical usage models can be used to generate test-cases automatically. Additionally, further integration into Mahr software is planned, including using the monitoring during actual alpha tests.

References

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[2] P. A. Brooks and A. M. Memon. Automated gui testing guided by usage profiles. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342, New York, NY, USA, 2007. ACM.

[3] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396 – 402, 1984.

[4] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31:187–202, 2005.

[5] Hewlet-Packard Company. HP Functional Testing, 2010.

[6] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2 of Special Sessions, page 1042, 1993.

[7] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 260, Washington, DC, USA, 2003. IEEE Computer Society.

[8] A. M. Memon. Gui testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, 2002.

[9] A. Nagarajan and A. Memon. Refactoring using event-based profiling. In *in Proceedings of The First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[10] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[11] A. Paiva, J. Faria, N. Tillmann, and R. Vidal. A model-to-implementation mapping tool for automated model-based gui testing. In *Formal Methods and Software Engineering*, volume 3785 of *LNCS*, pages 450–464. Springer, 2005.

[12] J. Sant, A. Souter, and L. Greenwald. An exploration of statistical models for automated test case generation. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[13] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A Capture/Replay tool for observation-based testing. *SIGSOFT Software Engineering Notes*, 25(5):158–167, 2000.

[14] P. Tonella and F. Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution*, 16(1-2):103–127, 2004.

[15] F. Willems, Y. Shtarkov, and T. Tjalkens. The context-tree weighting method: basic properties. *IEEE Transactions on Information Theory*, 41(3):653 –664, 1995.