GUI-Profiling for Performance and Coverage Analysis

Nico Beierle
Assystem Germany
Berlin, Germany
Email: nbeierle@assystem.com

Peter M. Kruse
Assystem Germany
Berlin, Germany
Email: pkruse@assystem.com

Tanja E.J. Vos
Open University, The Netherlands
Universitat Politecnica de Valencia, Spain
Email: tanja.vos@ou.nl, tvos@pros.upv.es

***Abstract*—Existing software analysis methods for performance and coverage are typically tied to the source code of software applications. In this work, we extend these methods to the Graphical User Interfaces (GUI) of applications, motivated by the desire to bring the user perspective into focus of software quality assurance and testing at the GUI level. We present and discuss various profiling procedures, their advantages and disadvantages, the arising challenges and the identified solutions. The identification and classification of the GUI components recorded during the monitoring process posed particular problems. The monitoring and collection of data could be well implemented, while detailed improvements in the evaluation of results are still necessary.**

I. Introduction

Profiling [1] is a dynamic analysis method that studies the behaviors of a software application during the execution phase. This allows analysts to obtain specific and detailed information of specific execution paths. This is in contrast to the static analysis where this information is obtained without execution from the source code. The corresponding tools, which enables analysis of the software through profiling, are called *profilers*.

In this paper we propose to use profiling at another level, namely the black-box testing of applications at the Graphical User Interface (GUI) level. The user experience, especially in the field of GUI applications, is an often underestimated quality feature in software development. The efficiency of an application as perceived by the user is generally differently from the perception of the developer [2]. The typical user only perceives the interaction with the user interface and accordingly insists on a fluent usability of all components.

Since the GUI of a software application targets the users, GUI testing should be done from a user's perspective and therefore intensify the testing of user aspects. Although the classical context of profiling is the source code, we propose that this can also be used when testing at the GUI level in a user-oriented development and testing environment.

Besides profiling, there is another type of metric that we want to borrow from source code analysis, namely coverage for GUI components. These coverage metrics will give us information on how well all aspects of the GUI have been exercised during our tests such that the profiling information can be used in context.

Coverage for GUI has not been extensively used and described in the literature so far, as far as we know there is only the work of Memon et al. [3]. However, especially when developing GUI tests, it is useful to determine a coverage rate for the events of the GUI components to provide the tester with an impression of the components' coverage rate.

In this paper, we describe our results of adapting, combining and optimizing the existing techniques for performance and coverage analysis into a prototype tool to be used when testing at the GUI level. This will result in test results that are more able to reflect the user's perspective and to improve the understanding of quality from a user's perspective.

Our paper is structured as follows. Section II describes our prototype implementation. Section III explains the concept of GUI coverage. Section IV then details GUI performance. Section V evaluates actual results and gives insights on the prototype implementation. Section VI finally concludes and presents future work.

II. Prototype

Our prototype tool for analyzing the GUI performance and coverage at the run time of a System Under Test (SUT) can basically be split up in three parts.

The first part implements the *monitoring* of the SUT to be analyzed. Here the coverage of the GUI components and the profiling information related to performance are collected (the details of these are described in the next sections). The application monitor is realized using a Java Agent.[1] Java Agents allow to instrument programs running on the Java Virtual Machine. For performance reasons we use ASM Java Bytecode manipulation.[2] JSON is then used as the transfer encoding [4] over a regular TCP connection to the server.

The second area provides the *client*, which represents the results of the monitoring activities during the tests. The application is implemented using the Angular framework[3] as a single-page application (SPA). Node.js, the JavaScript-based platform for the operation of network applications, is used for the realization of the web server (client server) offering a REST (Representational State Transfer) API.

The third and central component of the tool takes care of the *database* permanent storage and provision of the collected data. This component is realized in terms of a PostgreSQL database. Due to the existence of many different database concepts [5] it out of our current focus to make an extensive

[1]Java Agent: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html

[2]ASM Java Bytecode Manipulation: http://asm.ow2.org

[3]Angular framework for Single page Applications: https://angular.io

database comparison to find the optimal method for storing the information.

Both the monitoring-part and the client-part for the representation of results come with a *server* component which provides access to the database. These components are called *monitoring-server* and *client-server* respectively. The task of these two server components is to hide the database structure away from the client and the application monitor.

III. GUI Coverage Analysis

In order to analyze the GUI coverage while testing, we need to define the coverage criteria and subsequently determine how we can obtain the information to measure that metric.

The coverage analysis is carried out during the execution of the GUI tests, so that the degree of completeness of these tests can be determined. Consequently, the goal is to determine the coverage of the all GUI components, which are actively used during the tests. Like the well-known code coverage criteria, which calculates the coverage of the program constructs (i.e. statements, branches, paths), the GUI coverage criteria determines the covered graphical components of an application, the so-called widgets (i.e. buttons, labels, text fields, scroll-bars, menus).

The GUI of an application usually consists of many components (widgets), which can also be nested. This hierarchical structuring is referred to here and hereafter as a *widget tree*. The root node of such a tree is, in many cases, the application window itself, whereas the leaf nodes often consist of widgets which do not allow further child elements. The most frequently used leaf components are buttons, labels or input fields.

The structure of the widget tree is kept as simple as possible in order to be able to easily determine not only the entire covering, but also the coverage for each sub-tree. Each sub-tree forms a specific area of application.

We create the widget structure dynamically at the SUT level by logging and analyzing the application's behavior during the program run-time. We obtain the necessary information through the adaptation of the SUT at byte-code level, this avoids changing the source code and eliminates the need for new compilation. In language environments such as the Java Runtime Environment (JRE), byte-code adaptation is relatively simple and already provided. The JRE supports the implementation of custom class-loaders making it possible to directly influence the byte-code of each Java class, even before it is loaded into the JVM [6]. Profiling instructions supplement Java classes to evaluate the program behavior later.

Dynamic widgets of an application can be created or discarded at any time. Consequently, the structure of the widget tree keeps changing and needs to be updated continuously. Continuous logging provides an accurate image of the widget structure at all times. The overhead generated by the logging is kept as small as possible in the program, since logging only takes place exactly when changes occur in the widget structure.

Maintaining a dynamic widget tree, there will be similar widgets which have been instanced different times. These should appear in the tree only once, in order to be able to correctly determine the cover space during coverage analysis. For the allocation of the same widgets, a clear *identifiability* is an important prerequisite for the calculation of a meaningful result in the coverage analysis. We use a combination of different widget features for the correct determination of the equality [7], [8]. We use class name, call stack, attributes, and sub widgets in our approach.

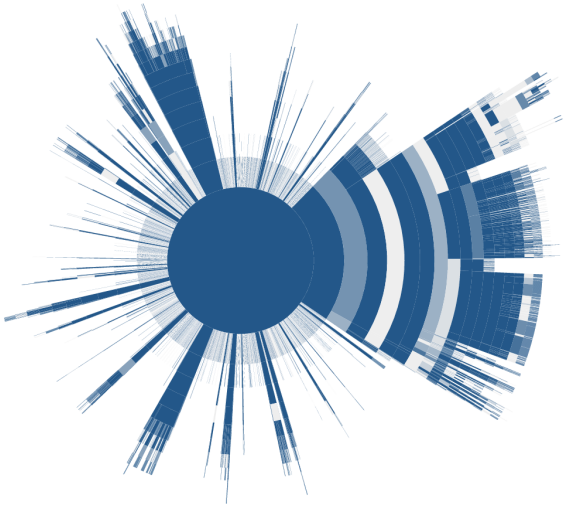


Fig. 1. Example of sunburst diagram representing Test Coverage

In general, only those GUI widgets, which have been instanced at least once during program execution, can be recognized in this method. To reach as many as possible, we have the analysis tool actively clicking through the GUI, in order to be able to instantiate all displayable widgets and to build up a complete widget tree [9].

A coloured widget tree can be used to represent the results, starting with the root node, and navigate down to individual leaf nodes. We call this a covering tree with colouring according to whether a widget was covered or not during the tests. Such a tree would take up a lot of space and make a cumbersome representation for investigation. Instead, we use a *sunburst* diagram (see Figure 1 for an example). The centre point marks the root node in this representation, and the outer regions characterize the leaves of the tree structure. Through the colour marking of the segments, this diagram can be enriched with further information. The advantage of this representation is that the complete coverage can be estimated at a glance. Partial trees with a low or particularly high covering rate can be identified quickly. A disadvantage of this representation is, however, that the large number of leaf nodes, depending on the depth of the tree, only fill a relatively small part of the representation and thus can lead to a deceptive assessment of an almost completely dark coloured diagram.

A second way to represent results of the coverage analysis are overlays on the application (see Figure 2 for an example). It allows a direct visual link between the GUI and the results of the coverage analysis and is best combined with the manual creation of Capture & Replay tests.

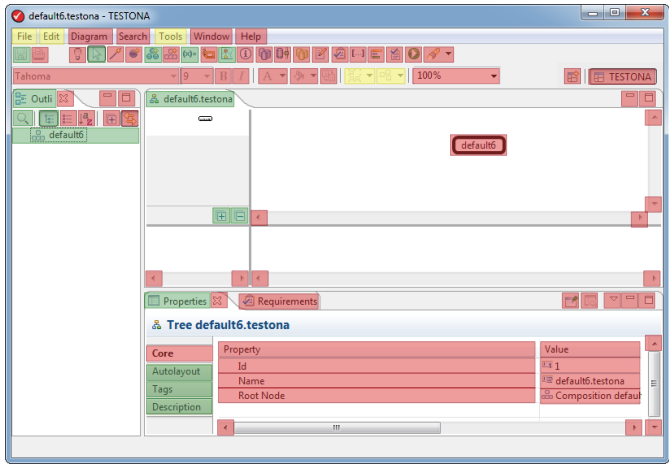
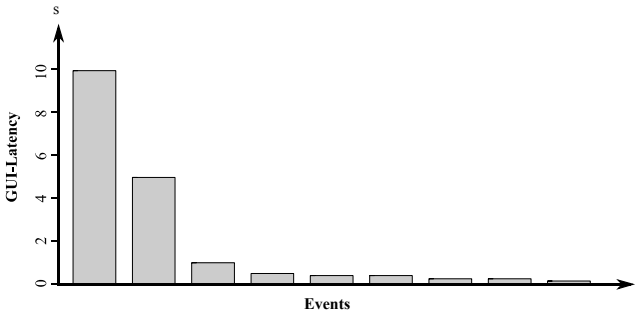


Fig. 2. Representing Coverage as GUI Overlay



Component	Event	Latency
Button (Cancel) ← Composite ← Dialog (Save)	selection	**10s**
Button (Save) ← Composite ← Dialog (Save)	selection	**5s**
Label (Filename) ← Composite ← Dialog (Save)	mouse.up	**1s**
Button (Generate) ← Composite ← Dialog (Test Generator)	selection	**511ms**
Text (...) ← Group ← Composite ← Dialog (Test Generator)	key.down	**392ms**

Fig. 3. Results of Performance Analysis

IV. GUI Performance Analysis

The GUI performance analysis aims to significantly expand the existing performance tests. As with the GUI Coverage analysis, these performance tests are also intended to refer to the widgets of the program interface. The basic idea is to link the performance tests with the functional GUI tests. In this interplay, each user event on the GUI triggered by the GUI test is simultaneously a performance test, for example by determining the run times of the methods executed by the event. With respect to the run times, particularly noticeable measured values (long running times) can be indicated in the performance evaluation.

The resource management of the individual applications is performed by the operating system in most modern system architectures. The administration and execution of the application stubs is controlled by the operating system so that the resource consumption can also be determined separately for each of these threads. These operating system run-time information provided by the operating system can also be used for the GUI performance analysis. In such a concept, the goal is first to synchronize the data from the monitoring of the program inputs with the run-time information of the operating system for resource consumption. Perhaps the only and most important characteristic of synchronization is time.

On the application level, it is possible to perform the performance measurements on the basis of the handler method performed by the event. These handler methods are here and in the following also to be referred to as event listeners. On the basis thereof, e.g. the method run time of the treatment routine can easily be determined.

For performance analysis, it is important that the measured values are displayed in a suitable form. This presentation is intended to separate the interesting ones from the less interesting results of the measurements, or to give an overall impression of the application performance. Such measured values, which exceed a certain defined limit value, are regarded as interesting (Figure 3).

V. Evaluation

Since the monitoring of the application behavior is compulsory performed on the same system and, in the case of implementation as a Java agent, even in the same application code, effects on the program behavior are unavoidable. The ultimate goal in monitoring must therefore be to minimize this influence, but in no case to change the program logic.

The application for evaluation is *TESTONA* (http://www.testona.net/), the graphical editor [10] for the classification tree method [11], a black box test design technique.

The existing test suite of TESTONA has a meaningful amount of 370 test cases [12]. Since not all tests of this test suite can be considered stable, the average number of failed tests is determined by the repeated execution of all tests. In addition to the number of failed tests, the run-time of the entire test execution is also logged. This process is then repeated with an activated monitoring.

Experiments with the 10-times repetitive test executions, each with monitoring on and off, show that the number of failed tests, as well as the run-time of the test execution, almost doubled with activated monitoring. While the doubling of the run-time can be directly attributed to the monitoring, the doubling of the error rate must be considered more critically. What is striking here is the much wider spread of the number of failed tests compared to the test without active monitoring. This indicates that the increased number of errors can be attributed to the less robust test creation. It is striking that the selection of the failed tests during test execution is very similar to the monitoring, because often the same tests fail. In the test execution with active monitoring, however, the lists of the failed tests differ significantly more. In conclusion, it is assumed that the monitoring process has a strong influence on the program run-time, but not the program logic. The higher number of failed tests can be justified by the extended execution time and the lower robustness of the tests.

In addition to the functional test assurance of the coverage analysis, the test coverage of the Capture & Replay tests was

analyzed. Figure 1 shows the graphical representation of the event coverage. The calculated test coverage is 8%.

When looking closely at the coverage tree, it is noticeable that many identical widgets are listed several times in the tree. The reason for this is the inadequate correct identification of the same widgets. We compared the characteristics of individual GUI widgets and found the very same error pattern in most samples: The cause of the error is the call stack (included in the comparison of widgets) differs in function calls, probably caused by the use of Java reflection.

For the verification of the performance analysis, a program component is selected in TESTONA, which generates a significantly increased GUI latency when it is used. During the test execution this GUI latency is to be provoked as the only large latency. In the expected test result this latency should be clearly visible in the performance representation in the client.

From the test result it can be seen that the largest GUI latency is caused by the selection of the corresponding menu entry. The determined 64 seconds indicate that the program interface is not accessible within this time span. The result thus corresponds to the expected result from the test specification.

The evaluation showed that, as expected, the monitoring process negatively impacts the program running time, but does not have a demonstrable effect on the program logic. On the basis of TESTONA the functionality of the monitoring and the results presentation was checked. The integration of the monitoring into the application was possible.

When evaluating the performance analysis, a long GUI latency was provoked as described above. The subsequent application of both test specifications confirmed the viability of the analytical procedures.

VI. Conclusion

The monitoring tool currently supports only the logging of widgets from the SWT or JFace framework. If monitoring is also to be provided for other Java-specific GUI frameworks, this functionality can be complemented by the monitoring tool. In this case, it is only required to specify the methods for the necessary code injection. However, it is necessary to create a separate monitoring tool to support monitoring of non-Java applications. Depending on the target environment, this monitoring tool can be similar to the implementation of the prototype: Functions for logging the status information can be injected directly into the application code or else the external interfaces of the runtime environment.

The reliable identification of GUI widgets is both essential and difficult to implement, not only in the GUI analysis methods presented here, also automated testing tools need to solve this problem [13].

For the Coverage Analysis, the prototype implemented in this work provides the coverage rates for simple event coverage. However, the coverage criterion can also include event chains in the analysis [3]. This would make it possible for the events to be viewed in a context. In such a context, it makes a difference, for example, which export format was selected before clicking the *Save* button. The monitoring in the current implementation has already recorded all events and their occurrences, so the executed event transitions can already be gathered relatively easily. However, to calculate the coverage, we need to know the size of the entire space, i.e. the set of actually possible event chains.

In addition to GUI latencies, a performance analysis of the background processes can also provide useful information for the quality assurance of the software. In particular, splitting the calculations to multiple threads and using thread pools makes this capture more difficult. It is therefore recommended to mark the corresponding threads activated by a GUI event recursively in order to enable allocation of the resource consumption. Future work will take up this idea and make further suggestions on how the resource consumption of an application can be mapped to the GUI events.

While the performance analysis focuses primarily on determining the latencies, other performance features are also of interest. In particular, the memory consumption. However, in contrast to the run-time analysis, the use of memory resources can be much more difficult, especially if these values are to be mapped to the individual GUI events.

Other future work consists of integrating the profiling and performance analysis with automates traversal based GUI test tools like TESTAR [14], so we can study the power of random or search-based testing. Additionally, the representation of the results in the client is so far only very rudimentary and can be improved or expanded in many places.

References

[1] D. Jackson and M. Rinard, "Software analysis: A roadmap," in *Conference on the Future of Software Engineering*. ACM, 2000, pp. 133–145.

[2] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang, "Hang analysis: fighting responsiveness bugs," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 177–190.

[3] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 256–267, 2001.

[4] JSON. Json. [Online]. Available: http://www.json.org

[5] D. Kroenke and D. J. Auer, *Database concepts*. Prentice Hall, 2010.

[6] C. Mcmanis. (1996) The basics of java class loaders. [Online]. Available: http://www.javaworld.com/article/2077260/learn-java/learn-java-the-basics-of-java-class-loaders.html

[7] K. Li and M. Wu, *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons, 2006.

[8] O. Stadie and P. M. Kruse, "Closing gaps between capture and replay: Model-based gui testing," in *1st INTUITEST Workshop*, 2015.

[9] A. M. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing." in *WCRE*, vol. 3, 2003, pp. 260–269.

[10] P. M. Kruse and M. Luniak, "Automated test case generation using classification trees," *Software Quality Professional*, vol. 13, no. 1, 2010.

[11] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[12] A. Kresse and P. M. Kruse, "Development and maintenance efforts testing graphical user interfaces: a comparison," in *7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM, 2016, pp. 52–58.

[13] S. Bauersfeld, T. E. J. Vos, N. Condori-Fernandez, A. Bagnato, and E. Brosse, "Evaluating the testar tool in an industrial case study," in *Proc. of the 8th ESEM*. ACM, 2014, pp. 1–9.

[14] T. E. J. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *IJISMD*, vol. 6, no. 3, pp. 46–83, 2015.