Performance Testing of GUI Applications

(Position Paper)

Milan Jovic, Matthias Hauswirth
Faculty of Informatics
University of Lugano
Switzerland
(Milan.Jovic,Matthias.Hauswirth)@usi.ch

***Abstract*—Current GUI testing approaches validate the functional correctness of interactive applications, but they neglect an important non-functional quality: performance. With the growing complexity of interactive applications, and with their gradual migration to resource-constrained devices such as smartphones, their performance as perceived by the human user is growing in importance. In this paper we propose to broaden the goal of GUI testing to include the validation of perceptible performance in addition to functional correctness.**

***Keywords*-interactive applications; performance testing; record/replay; latency**

I. Introduction

A large segment of today's computer programs are interactive applications with graphical user interfaces. These applications are written in an event-based style, where the application needs to handle a diverse set of events representing user inputs. A significant body of work is concerned about methodologies for testing the correct behavior of such GUI applications. Existing approaches usually play previously recorded event sequences [1], [2], [3], [4], programmatically generated event sequences [5], or event sequences generated from a model [6], [7], [8], [9], [10], to automatically test the GUI of an interactive application.

In this position paper we would like to broaden the applicability of that work from its focus on functional correctness to also include a non-functional property: perceptible performance. Perceptible performance is an important quality of interactive software. As Wang et al. [11] state, the Microsoft Office development team has conducted a survey where they found that user complaints about bad perceptible performance are almost as frequent as complaints about crashes.

The perceptible performance of an interactive application can be characterized by its latency of handling user events [12]. As Shneiderman has shown, event handling latency can induce frustration, annoyance, and anger [13], and it can significantly affect user productivity [14]. He found the threshold of perceptibility to lie around 100 ms. MacKenzie and Ware [15] found that for latencies up to 225 ms, user performance in virtual reality environments significantly decreased with increasing latency, and Dabrowski and Munson [16] found specific perceptibility thresholds for keyboard input (150 ms) and mouse input (195 ms).

Given this impact that *functionally correct* event handlers with a perceptibly long latency can have on users of a GUI application, we believe that GUI testing methodologies should include performance testing as one of their goals.

Unfortunately, extending functional GUI testing methodologies to support performance testing is non-trivial. First, a GUI performance testing approach cannot just measure the time it takes to complete a test case, but it needs to measure performance as perceived by a human user. This involves measuring the latencies of handling individual user events, and, to support debugging, gathering information about the causes of long latencies. Moreover, the test infrastructure has to have a low-enough overhead to not perturb the measured latency. Second, the event handling latency can differ greatly between the minimal interactions we seek for efficient functional testing and the realistic interactions occurring in the field. Performance problems in interactive applications become particularly visible for long interaction sequences, on large documents, in complex applications.

Despite these complications, we believe that existing functional GUI testing approaches can be enhanced to effectively support GUI performance testing. In this paper we characterize the GUIs of a set of real-world Java applications, demonstrating their complexity, we discuss the factors affecting the performance of such complex interactive Java applications, we outline the requirements those factors impose upon an effective GUI performance testing approach, and we present a research agenda leading towards such an approach.

The remainder of this paper is structured as follows: Section II discusses listener latency profiling, a means to measure a GUI application's perceptible performance. Section III describes causes of performance bugs in GUI applications. Section IV characterizes the complexity of modern Java GUI applications and discusses how this complexity affects performance testing. Section V presents the requirements for an automated GUI performance testing approach, Section VI outlines a research agenda leading towards such an approach, and Section VII concludes.

II. Measuring GUI Performance

In prior work [12] we have developed "listener latency profiling", an approach to measure the event handling latency of arbitrary Java GUI applications. LiLa, our

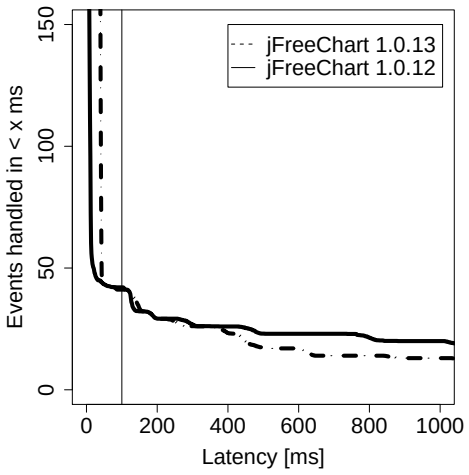

Figure 1. Latency distributions of two versions of JFreeChart

listener latency profiler implementation, performs binary rewriting of Java classes to instrument the event dispatch code. The instrumentation consists of code to capture the current time, and to generate a trace of the measured latencies. These traces contain information about the latency of each GUI event handled by the application. We use the cumulative distribution of event handling latencies to quantify the overall responsiveness of the application.

Figure 1 shows such a cumulative latency distribution. The x-axis shows the latency in milliseconds. The y-axis shows the number of events that were handled in less than x milliseconds. The vertical line at 100 ms represents Shneiderman's perceptibility threshold. An ideal curve would have the shape of an "L": it would drop to 0 at an $x < 100$ ms, which would mean that none of the events incurred a perceptible latency.

The two curves in Figure 1 represent two different versions of a Java application for visualizing time charts using the JFreeChart[1] library. The old version of JFreeChart, 1.0.12, includes a performance bug, which significantly slows down the rendering of time charts. This bug was fixed in version 1.0.13, and the corresponding curve shows a significant improvement in perceptible performance: there are fewer episodes longer than 400 ms, and more episodes shorter than 60 ms.

III. Causes of GUI Performance Bugs

Why are interactive applications perceptibly slow? The most obvious reason is an inefficient design or implementation of the application. While the application's event handlers themselves certainly can cause perceptible latency, e.g. because they synchronously access remote data, or because they use inefficient approaches to render complex visual output, components *outside* the event handlers, or even outside the application, can drastically affect perceived performance as well [17].

Many modern GUI applications are written in managed languages such as Java. These languages are based on powerful runtime environments that provide features such as reflection, dynamic class loading, just-in-time compilation, and automatic memory management. While these rich services can significantly improve programmer productivity, they also incur a cost in terms of runtime overhead, and thus a potential reduction of perceptible performance.

For example, dynamic class loading significantly limits the precision of static analysis of application code, and thus severely constrains the applicability of compiler optimizations. To compensate for these limitations, modern virtual machines often perform optimizations in a speculative way (e.g. speculative inlining), and they have to undo those optimizations when certain constraints are violated (e.g. they have to outline an inlined method when a subclass overriding that method gets loaded). Such potentially costly "deoptimizations" can happen in an interactive session at times that are not obvious to an application programmer (e.g. when the activation of a seemingly unrelated plugin triggers the loading of some classes).

A more straightforward example is the latency due to garbage collection (GC). An event handler may incur significant latency when it triggers a GC. While such a collection is *triggered* by the handler (due to the handler's code allocating some object when there is not enough free space in the heap), it may not be the handler's "fault" that the heap was full. A completely unrelated piece of application or library code may have caused an excessive amount of object allocations, causing the heap to fill up.

Another possible cause for long latency events lies in the increasing concurrency needed to exploit the multicore processors in modern desktop computers. GUI applications are mostly single-threaded, where one GUI thread dispatches events from a GUI event queue. In order to improve the performance of complicated GUI events (such as the query of a database in response to a click on a search button), developers often perform long-running computations in additional threads. This model requires the communication between threads, and the corresponding synchronization. A suboptimal design or implementation of such background activities can, for example, lead to lock contention, and can significantly slow down the GUI thread.

In conclusion, while the causes of GUI performance bugs sometimes are located directly in the event handlers implemented by the application (and thus may occur even when a specific event is replayed out of context), performance bugs in modern complex GUIs can lie in places that are not directly related to the handler of a long-latency event. This means that an approach to GUI performance testing needs to consider realistic interactive sessions, with sequences of interactive events as they would occur in a use of the application in a production scenario.

[1] http://www.jfree.org/jfreechart/

IV. Characterization of Java GUI Applications

Many modern GUI applications are complex software systems, with large numbers of GUI components and events, which makes testing their performance non-trivial. Application size impacts testability because of the interactions between the many components through the global state of the application. In case of performance testing, this problem is emphasized by the fact that components also interact indirectly through the lower layers of a computer system (e.g. through the mechanisms described in Section III).

Application	Description
Arabeske 2.0.1	Arabeske pattern texture editor
ArgoUML 0.28	UML CASE tool
CrosswordSage 0.3.5	Crossword puzzle editor
Euclide 0.5.2	Geometry construction kit
FreeMind 0.8.1	Mind mapping editor
GanttProject 2.0.9	Gantt chart editor
jEdit 2.7pre3	Programmer's text editor
JFreeChart Time 1.0.13	Chart library, showing temporal data
JHotDraw Draw 7.1	Vector graphics editor
Jmol 11.6.21	Chemical structure viewer
LAoE 0.6.03	Audio sample editor
NetBeans Java SE 6.5.1	Integrated development environment

Table I
Applications

For this reason we quantify the complexity of a set of real-world Java GUI applications. We have chosen the twelve applications in Table I because they are open source, and because of their rich interaction styles. In particular, we favored applications with a direct-manipulation style, where the user can directly interact with a visual representation (e.g. a diagram, or a sound wave) of the application's data. These kinds of applications are often more performance-critical than simple form-based programs where users just select menu items, fill in text fields, and click buttons. We use CrosswordSage, FreeMind, and GanttProject because they have been selected as subjects in prior work on GUI-testing [7] based on their active developer community and high all-time activity scores on SourceForge. We completed our suite of programs by picking realistic and useful applications from a wide range of application domains.

Application	Classes	Components	Windows	Listeners
Arabeske	222	41	4	76
ArgoUML	5349	347	43	1431
CrosswordSage	34	10	3	15
Euclide	398	33	16	66
FreeMind	1909	36	6	162
GanttProject	5288	152	22	413
jEdit	1150	175	34	265
JFreeChart	1667	368	291	455
JHotDraw	1146	86	7	409
Jmol	1422	50	17	156
LAoE	688	68	8	204
NetBeans	45367	2019	37	6264

Table II
Static Characteristics of GUI Applications

Table II characterizes the chosen applications. Column "classes" shows the number of application classes. It excludes the classes of the Java runtime library. Column "components" shows the number of GUI components. The values in this column essentially represent the number of GUI element types. This includes all subclasses of `java.awt.Component`, and thus includes all AWT and all Swing component subclasses implemented in the application, including top-level components such as frames and dialogs. Column "windows" shows the number of application classes that extend `java.awt.Window`, which includes application subclasses of all AWT and Swing top-level components (such as dialogs and frames). Column "listeners" shows the number of application classes that implement `java.util.EventListener`. These classes essentially represent the different event handlers. They include the implementations of all AWT and Swing listener interfaces.

We can see that the complexity of GUI applications varies widely. The simplest application, CrosswordSage, consists of only 34 classes. The most complex application, NetBeans, is over 1000 times larger. While NetBeans represents an outlier, it also represents the probably most widely used application in our sample. Testing a realistic application like NetBeans will push the boundaries of any GUI testing approach, and in particular any potential GUI performance testing approach.

Table II shows that the applications consist of between 10 and 2019 component classes. Note that this number represents a count of component *types*, and it excludes the runtime library classes (such as `JButton`). A running application may contain multiple instances of many of these component types. The number of window subclasses ranges between 3 and 291. Windows represent top-level components such as the main frame and the various modal dialogs. A typical application only contains a small number of such classes. With a count of 291, JFreeChart contains an abnormally large number of window subclasses (79% of components). The reason for this is that JFreeChart is essentially a library, not a single application, and that it contains a large number of demonstration applications, each of them with its own window.

V. Requirements for GUI Performance Testing Approaches

Given the issues discussed in the previous sections, a GUI *performance* testing approach needs to fulfill the following specific requirements:

Realistic sessions. Given the significant complexity of many modern GUI applications, and given the indirect performance interactions between otherwise unrelated software components, the interactive session executed for performance testing needs to represent a realistically long sequence of events. Moreover, the inputs (e.g. the number and sizes of documents edited in an application) have to be of a realistic size, so that the algorithms used in the event handlers get exercised to a realistic degree.

Minimal perturbation. A session replay tool must not affect the perceptible performance of the application.

This constraint is crucial to prevent the perturbation of the measured event handling latency. If, for example, the replay tool allocates a significant number of objects, then the execution of an event handler may be perturbed by garbage collections that are to some degree caused by the replay tool.

The above requirements add to the requirements for functional GUI testing approaches. In some cases they may also conflict with those existing requirements. For example, the "GUI element identification problem", attacked by McMaster and Memon [18], may require solutions that introduce significant overhead (e.g. by requiring interaction scripts to carry additional information about each GUI element, or by requiring more complex traversals of the component tree to find a given GUI element).

VI. Research Agenda

We identified the performance bug in JFreeChart, shown in Figure 1, by manually repeating the same complicated interactive session on the two different versions of the application. We repeated this process multiple times to ensure that the difference between latency distributions was not accidental. That approach requires a significant human effort and clearly is not scalable in practice.

Our goal is to *automate* GUI *performance* testing to make it practical. To reach that goal, we will need to find an approach to replay realistic interaction sequences on the large interactive applications that most depend on performance test automation.

We first plan on investigating the suitability of existing functional GUI testing tools for GUI performance testing. Based on these results we will need to devise improvements over the existing tools, and to resolve possible conflicts between requirements, especially in terms of robustness of GUI element identification vs. perturbation.

We plan to address the following questions:

1) To what degree do existing record/replay GUI testing tools, such as Abbot [1], Jacareto [2], Pounder [3], Marathon [19], or JFCUnit [4] fulfill the GUI performance testing requirements? In particular, how do different GUI element identification approaches fare in the trade off between robustness and performance perturbation?
2) How can the robustness of record/replay tools be improved? What aspects of GUI element identification are most relevant for long realistic interaction sequences? What other aspects of GUI record/replay significantly impact robustness? To what degree does non-determinism (due to multi-threading or due to changes in the environment) affect the robustness of GUI record/replay approaches?
3) How can the performance perturbation due to record/replay tools be reduced? Could some of the processing that traditionally takes place during the replay phase (and thus can affect replay performance) be moved to the recording phase, or even an offline phase?
4) To what degree is it possible to use model-based GUI testing approaches such as those based on event-flow graphs [6] for performance testing? Do the model-based tests executed by such tools stress (in terms of performance) the same aspects of the application as a realistic trace? Does model-based GUI performance testing require new kinds of coverage criteria?
5) Do automatic GUI test case repair tools [20] scale to the long event sequences needed for GUI performance testing of applications like NetBeans?
6) What is the best approach to determine whether the change between two cumulative latency distributions represents a significant performance bug?
7) Is the cumulative latency distribution a reliable measure of perceptible performance? For example, what about bursts of interactive events, where each individual event is handled quickly, but the entire burst takes a perceptible amount of time?
8) To what degree is perceptibility correlated to user satisfaction? If a user anticipates a specific event (e.g. the response to pressing the "Compile" button in an IDE) to take a long time, he may be more tolerant and even accept a significant perceptible latency. Is it possible to automatically associate different acceptable latencies to different GUI events?

VII. Conclusions

In this position paper we broaden the notion of GUI testing to include GUI performance testing. We outline the challenges specific to performance testing of GUI applications, and we propose a research agenda towards an automatic approach that would make GUI performance testing practical. We are very interested in interacting with the GUI testing community and to learn about their perspective on our thoughts, and we hope to contribute to the GUI testing community with our experience on performance measurement of interactive applications.

References

[1] T. Wall, "Abbot framework for automated testing of java gui components and programs." [Online]. Available: http://abbot.sourceforge.net/

[2] C. Spannagel, "Jacareto: A capture & replay tool for programs written in Java." [Online]. Available: http://jacareto.sourceforge.net/

[3] M. Pekar, "Pounder: Java GUI Testing Utility." [Online]. Available: http://pounder.sourceforge.net/

[4] M. Caswell, V. Aravamudhan, and K. Wilson, "jfcUnit: An extension to JUnit." [Online]. Available: http://jfcunit.sourceforge.net/

[5] A. Iline, "Jemmy: A Java UI Testing Tool." [Online]. Available: https://jemmy.dev.java.net/

[6] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of gui test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, pp. 884–896, 2005.

[7] 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*. New York, NY, USA: ACM, 2007, pp. 333–342.

[8] Y. Sun and E. L. Jones, "Specification-driven automated testing of gui-based java programs," in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. New York, NY, USA: ACM, 2004, pp. 140–145.

[9] A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal, "Modeling and testing hierarchical guis," in *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.

[10] A. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A model-to-implementation mapping tool for automated model-based gui testing," in *Proceedings of the 7th International Conference on Formal Engineering Methods*, 2005, pp. 450–464.

[11] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang, "Hang analysis: fighting responsiveness bugs," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA: ACM, 2008, pp. 177–190.

[12] M. Jovic and M. Hauswirth, "Measuring the performance of interactive applications with listener latency profiling," in *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*. New York, NY, USA: ACM, 2008, pp. 137–146.

[13] B. Shneiderman, *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, 1986.

[14] ——, "Response time and display rate in human performance with computers," *ACM Comput. Surv.*, vol. 16, no. 3, 1984.

[15] I. S. MacKenzie and C. Ware, "Lag as a determinant of human performance in interactive systems," in *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1993.

[16] J. R. Dabrowski and E. V. Munson, "Is 100 milliseconds too fast?" in *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, 2001.

[17] A. Adamoli, M. Jovic, and M. Hauswirth, "LagAlyzer: A latency profile analysis and visualization tool," in *ISPASS '09: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[18] S. McMaster and A. M. Memon, "An extensible heuristic-based framework for gui test case maintenance," in *TESTBEDS '09: Proceedings of the First International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*, 2009.

[19] D. Karra, "Marathon." [Online]. Available: http://www.marathontesting.com/

[20] A. M. Memon, "Automatically repairing event sequence-based gui test suites for regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–36, 2008.