Automating Performance Testing of Interactive Java Applications

Milan Jovic, Andrea Adamoli, Dmitrijs Zaparanuks, Matthias Hauswirth
Faculty of Informatics
University of Lugano
milan.jovic|andrea.adamoli|dmitrijs.zaparanuks|matthias.hauswirth@usi.ch

ABSTRACT

Interactive applications with graphical user interfaces are prevalent in today's environment: Everybody with access to any kind of computer constantly uses them. A significant body of prior work has devised approaches for automating the functional testing of such applications. However, no such work exists for automatically testing their *performance*. Performance testing imposes additional requirements upon GUI test automation tools: the tools have to be able to replay complex interactive sessions, and they have to avoid perturbing the application's performance. We study the feasibility of using five Java GUI capture & replay tools for GUI performance test automation. Besides confirming the severity of the previously known *GUI element identification problem*, we also identify a related problem, the *temporal synchronization problem*, which is of increasing importance for GUI applications that use timer-driven activity. We find that most of the tools we study have severe limitations when used for recording and replaying realistic sessions of real-world Java applications, and that all of them suffer from the temporal synchronization problem. However, we find that the most reliable tool, Pounder, causes only limited perturbation, and thus can be used to automate performance testing. Besides the significance of our findings to GUI performance testing, the results are also relevant to capture & replay-based functional GUI test automation approaches.

1. INTRODUCTION

In this paper we study whether it is practical to automatically test the performance of interactive rich-client Java applications. For this, we need to address two issues: (1) we need a metric and a measurement approach to quantify the performance of an interactive application, and (2) we need a way to automatically perform realistic interactive sessions on an application, without perturbing the measured performance.

We address the first issue by by measuring the distribution of system response times to user requests [5]. The second issue is the key problem we study in this paper: Instead of employing human testers who repeatedly perform the same interactions with the application, we evaluate different approaches to record an interactive session once and to automatically replay it the required number of times.

GUI test automation is not a novel idea. However, we are not aware of any automatic GUI performance testing approach that can evaluate the *performance* as *perceived by the user*. This kind of GUI performance test automation has two key requirements that go beyond traditional GUI test automation: (1) the need to replay realistically complex interactive sessions, and (2) the minimal perturbation of the measured performance by the tool.

First, many existing GUI test automation approaches and tools primarily focus on functional testing, and thus do not need to support the capturing and replaying of realistically long interactive sessions. However, for *performance* testing, the use of realistic interaction sequences is essential. The reasons for this are based on the problem that applications interact with the underlying platform in non-functional ways, and that these interactions can significantly affect performance. For example, excessive object allocations in one part of an application may indirectly trigger garbage collection during the execution of a different part; or the first use of a class may trigger dynamic class loading, may cause the language runtime to just-in-time compile and optimize application code, and may even cause previously optimized code in a different class to be deoptimized. Finally, the size of data structures (e.g. the documents edited by the user) directly affects performance (the runtime of algorithms depends on data size), but it can also indirectly affect performance (processing large data structures decreases memory locality, and thus performance). To observe these direct and indirect effects, which can be significant when programs are run by real users, we need an approach that replays realistically complex interaction sequences.

Second, existing GUI test automation tools are not constrained in their impact on performance. However, to allow GUI *performance* test automation, a tool must not significantly perturb the application's performance. Record & replay tools can cause perturbation due to additional code being executed (e.g. to parse the file containing a recorded session, or to find the component that is the target of an event), or additional memory being allocated (e.g. to store the recorded session in memory). Thus, we need a capture & replay approach that incurs little overhead while still being able to replay realistically complex sessions.

In this paper we evaluate capture & replay approaches as implemented in a set of open-source Java GUI testing tools.

AST '10, May 3-4, 2010, Cape Town, South Africa

We study the *practicality* of replaying complete interactive sessions of real-world rich-client applications, and we quantify the *perturbation* caused by the different tools.

While we specifically focus on performance test automation, to the best of our knowledge, our evaluation also constitutes the first comparative study of GUI test automation tools with respect to functional, not performance, testing.

The remainder of this paper is structured as follows. Section 2 describes our approach for characterizing interactive application performance. Section 3 introduces GUI capture & replay tools. Sections 4 and 5 evaluate the practicality of these tools for replaying small, isolated GUI interactions as well as complete sessions with real-world applications, and Section 6 discusses our results. Section 7 studies how the capture & replay tools perturb our measurements. Section 8 discusses threats to the validity of our study, Section 9 summarizes related work, and Section 10 concludes.

2. GUI EPISODE DURATION AS PERFORMANCE MEASURE

To characterize the perceptible performance of interactive applications, we measure the response time of user requests (episodes) by instrumenting the Java GUI toolkit's event dispatch code. The instrumentation measures the wall clock time it took the application to handle the request. As research in human computer interaction has found, requests that are shorter than a given threshold (around 100 ms [5]) are not perceptible by the user. Requests longer than 100 ms are perceptible and can negatively affect user satisfaction and productivity. We thus collect a histogram of request lengths, which we represent as cumulative latency distribution plots in this paper.

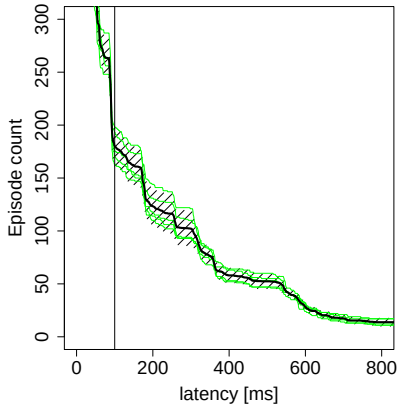

Figure 1: Cumulative Latency Distribution

Figure 1 shows an example of such a plot. The x-axis represents the latency in milliseconds. The y-axis shows the number of episodes that take *longer* than the given x ms. A vertical line at 100 ms represents the perceptibility threshold. Response times to the left of that line are not perceptible. An ideal curve would be L-shaped, where the vertical part of the L could lie anywhere between 0 and 100 ms, and the horizontal part should lie at 0, that is, there would be 0 episodes that took longer than 100 ms.

Given that it is impossible to accurately repeat the same user interaction multiple times with the exact same movements and timings, the cumulative latency distribution differs slightly between runs. Figure 1 shows multiple curves. The five thin lines represent the latency distributions for five runs of the same interactive session. The thick line represents the mean over these five distributions. Finally, the hatched pattern represents the confidence area for the mean. We computed the confidence area by computing a confidence interval for each point along the x-axis: e.g. we computed a 95% confidence interval for the five points at $x = 400$ ms, another one for the five points at $x = 401$ ms, and so on.

3. CAPTURE & REPLAY TOOLS

GUI capture & replay tools have been developed as a mechanism for testing the correctness of interactive applications with graphical user interfaces. Using a capture & replay tool, a quality-assurance person can run an application and record the entire interactive session. The tool records all the user's events, such as the keys pressed or the mouse movements, in a log file. Given that file, the tool can then automatically replay the exact same interactive session any number of times without requiring a human user. By replaying a given log file on a changed version of the application, capture & replay tools thus support fully-automatic regression testing of graphical user interfaces.

GUI capture & replay tools are usually not used for recording *entire* interactive sessions: their main goal is to record simple interaction sequences, such as the user clicking on the "File|Open..." menu, and to verify that this click indeed pops up the application's dialog window to open files. In this paper we study whether existing GUI capture & replay tools can be "abused" to record entire interactive sessions with complex real-world applications, and whether the tools allow or preclude the accurate measurement of perceptible performance given the overhead they impose on the application.

3.1 Tools used in this Study

Table 1 shows the five capture & replay tools we evaluate in this paper. All tools are written in pure Java and are available as open-source. They are all capable of recording and replaying interactive sessions of interactive applications based on Java's standard Swing GUI toolkit.

Tool	Version	Date	Status	Classes
Abbot	1.0.2	2008-08	active	577
Jacareto	0.7.12	2007-03	(active)	1085
Pounder	0.95	2007-03	dead	156
Marathon	2.0b4	2009-01	active	694
JFCUnit	2.08	2009-01	dead	150

Table 1: Capture & Replay Tools

Abbot[1] is a framework for GUI testing. Its basic functionality allows a developer to write GUI unit tests in the form of Java methods which call into the Abbot framework to drive an application's GUI. Besides tests written in Java, Abbot also allows the specification of tests in the form of XML test scripts. It provides a script editor, Costello, for editing such scripts. Besides the manual editing of test scripts, Costello also supports the recording of scripts by capturing the events occurring in a running application.

Jacareto[2] is a GUI capture & replay tool supporting the creation of animated demonstrations, the analysis of user

[1]http://abbot.sourceforge.net/
[2]http://jacareto.sourceforge.net/

behavior, as well as GUI test automation. Given this broad spectrum of applications, Jacareto provides a number of extra features, such as the highlighting of specific components in the GUI, extension points to capture and replay application-specific semantic events, or the embedding of Jacareto into the GUI application for providing macro record and replay functionality. Jacareto comes with two front-ends, CleverPHL, a graphical tool with extensive support for recording, editing, and replaying interaction scripts, and Picorder, the command-line record and replay tool we use in this paper.

Pounder[3] is exclusively focused on capturing and replaying interactions for GUI testing. It stores interaction scripts as XML files and is not intended to be used for manually writing tests. Compared to Abbot and Jacareto, Pounder is a lightweight tool, as can be seen by its narrow focus and its small size (number of classes in Table 1).

Marathon[4] seems to be an open-source version of a more powerful commercial product. Besides providing a recorder and a player, Marathon also comes with an extensive editor for interaction scripts. Marathon records interaction logs as Python scripts.

JFCUnit[5] is an extension that enables GUI testing based on the JUnit[6] testing framework. JFCUnit allows a developer to write Java GUI tests as JUnit test case methods. The main focus of JFCUnit is the manual creation of GUI tests (following JUnit's approach), but a recording feature has been added in a recent version.

3.2 Platform

We ran our experiments on a MacBook Pro with a Core 2 Duo processor. We used two different operating systems, Mac OS X and Windows. The version of OS X we used was 10.5.7. On top of it we ran Apple's Java 1.5.0_19_137 VM. One of our applications required Java 1.6, so in that specific case we used Apple's Java 1.6.0_13 VM. For the Windows-based experiments we used Windows 7 Beta 2, with Sun's Java 1.5_0.16 resp. 1.6.0_07 VM. We used the client configuration (-client command line option) of the virtual machine, because this is the option that is supposed to provide the best interactive performance.

4. REPLAY OF ISOLATED FEATURES

Before using the tools on real-world applications, we first evaluate them on a suite of simple test applications. Each of these test applications only exhibits a single relevant GUI feature. This is important, because if tools fail to properly capture & replay interactions of these fundamental features, it will be difficult to record and replay the rich and realistic interactions necessary for GUI performance testing of complete real-world applications.

4.1 Applications

Table 2 shows the eight test applications we developed for this purpose. They are divided into three categories: four tests for keyboard and mouse input, four widget tests (component events, scroll pane, file dialog, and combo box), and one timing test. Each test consists of a minimal GUI application that exposes the specific feature we want to test. The last test is special: it evaluates whether a capture & replay tool can faithfully maintain timing information. It provides a component that toggles between two colors, red and white, at a rate of 300 ms, driven by a timer. While recording, we click on the component only when it is red. We then count how many clicks the replay tool correctly performs during red, and how many it incorrectly delivers during white periods. Replay tools that fail this test will have difficulties replaying interactive sessions where users interact with animations (such as computer games).

Test	Description
TextField	Keyboard input in JTextField
MouseMove	Mouse movements
MouseDrag	Mouse drags (while button pressed)
MouseClick	Mouse button clicks
Component	Detection of component events
Scrolling	Scrolling a JTextArea
FileDialog	Navigating directory tree in JFileChooser
ComboBox	Selecting item in JComboBox popup
Timing	Synchronization of clicks with timer

Table 2: Test Applications

4.2 Results

Table 3 shows the results we obtained by running the five capture & replay tools on our test suite. The ✓symbol means that the test was successful, ✓* means that a minor part was failing, (✓) means that about the 50% of the test didn't work, but we could complete our task, and an empty cell means that we could not accomplish our task.

	Tool				
Test	Abbot	Jacareto	Pounder	JFC	M.thon
TextField	✓	✓	✓		
MouseMove	✓	✓	✓		
MouseDrag	(✓)	✓	✓		
MouseClick		✓	✓		
Component	✓	✓	✓		
Scrolling	✓*	✓*	✓		
FileDialog		(✓)	(✓)	✓	
ComboBox		✓	✓	✓	✓
Timing					

Table 3: Test Results

TextField. Abbot, Jacareto, and Pounder correctly record interactions with a text field, including entering and selecting text. JFCUnit and Marathon do not properly record text selections with the mouse.

MouseMove. Unlike the other tools, JFCUnit and Marathon do not record any mouse moves.

MouseDrag. Jacareto and Pounder properly record press, drag, and release events of a drag gesture. Abbot only records the release event, JFCUnit replays a drag as a move, and Marathon does not record drags at all.

MouseClick. Unlike the other tools, which fail to replay some buttons, Jacareto and Pounder correctly replay clicks with any of the three mouse buttons.

Component. JFCUnit and Marathon cannot replay interactions with a top-level component (a frame or a dialog). If a user moves or resizes a window, these actions are lost.

[3]http://pounder.sourceforge.net/
[4]http://www.marathontesting.com/
[5]http://jfcunit.sourceforge.net/
[6]http://junit.sourceforge.net/

Scrolling. Pounder is the only tool that supports autoscrolling (selecting text with the mouse and extending the selection past the viewport), scrolling using the mouse-wheel, by dragging the scroll bar knob, and by clicking. Abbot and Jacareto do not record mouse-wheel based scrolling. The other tools do not properly support scrolling.
FileDialog. Jacareto and Pounder do not support the selection of files with double-clicks. Abbot and Marathon do not properly work with file dialogs. JFCUnit is the only tool that supports all necessary operations.
ComboBox. Abbot has problems with heavy-weight popup windows and was unable to replay interactions with dropdown combo boxes.
Timing. None of the tools supports deterministic replay with respect to a timer.

Given the major limitations of JFCUnit and Marathon, we focus the remainder of the paper on Abbot, Jacareto, and Pounder.

5. REPLAY OF COMPLETE APPLICATIONS

Besides evaluating the capture & replay tools on simple tests, we also want to study the practicality of using them for recording sessions with real-world applications, and we want to evaluate to what degree they perturb the application performance at replay time.

5.1 Applications

Table 4 shows the twelve interactive applications we chose for our study. All applications are open source Java programs based on the standard AWT/Swing GUI toolkit. The table shows that our applications range in size from 34 classes to over 45000 classes (not including runtime library classes).

We chose applications from a wide range of application domains. We mostly focused on programs that provide rich interaction styles, where the user interacts with a visual representation (e.g. a diagram, or a sound wave). 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 [2].

5.2 Approach

To determine whether the different tools are able to capture and replay realistic interactions on real-world applications, we conducted the following experiment for each combination of <*tool, application, os*>, where *os* means either Windows or Mac OS X:

1. We studied the application to determine a realistic interaction script. The interaction script is a human-readable set of notes we use so we can *manually* rerun the same interactive session multiple times.

2. We ran the application under the capture tool to record the interaction.

3. If the capture failed, we either modified our environment or adjusted the interaction slightly, and went back to step 2.

4. We used the replay tool to replay the recorded interaction.

5. If the replay failed, we either manually edited the recorded interaction log and went back to step 4, or we modified our environment or adjusted the interaction slightly, and went back to step 2.

5.3 Interactive Sessions

We devised a realistic interactive session for each application and recorded it with each of the remaining capture tools (Abbot, Jacareto, and Pounder). We did this separately on each platform (Mac OS X and Windows), because we found that replaying a session on a platform different from where it was recorded failed in many situations, because the GUI's structure and layout between the two platforms can differ significantly.

We avoided features that, according to the results of Section 4, are not supported by the tools (e.g. double-clicks on FileDialogs). Where possible, we prepared documents of significant complexity (e.g. a drawing with 400 shapes for JHotDraw) ahead of time, and we opened and manipulated them during the sessions, using the various features provided by the applications.

5.4 Results

We now show which tools were able to properly capture and replay our interactions. **Table 5** presents one row for each application and two columns (Mac and Windows) for each tool. A check mark (✓) means that we were able to capture and replay the described interaction. A star (*) means we could record and replay a brief session, but that either we were unable to record a more meaningful session, or that we were unable to fix the recorded meaningful session so we could replay it. Finally, an empty cell means that we were unable to record any session of that application with the given tool.

Application	Tool					
	Abbot		Jacareto		Pounder	
	Mac	Win	Mac	Win	Mac	Win
Arabeske	*	✓	*	✓	*	✓
ArgoUML	*	*			*	✓
CrosswordSage	*	✓	✓	✓	✓	✓
Euclide	*	*	✓	✓	✓	✓
FreeMind		*		*	*	✓
GanttProject	*	✓	✓	✓	✓	✓
jEdit			✓	✓	✓	✓
JFreeChart Time	✓	✓	✓	✓	✓	✓
JHotDraw Draw	*	✓	*	✓	*	✓
Jmol	*	*	✓	✓	✓	✓
LAoE	*	*	*	*	✓	✓
NetBeans Java SE			✓	✓	✓	✓

Table 5: Tool Applicability

The table shows that Abbot and Jacareto had considerable limitations. They were unable to properly record and replay some of the required interactions. Moreover, Abbot does not record timing information, and thus is unable to replay a session log at the speed the user originally recorded it. An advantage of Abbot and Jacareto that does not show in the table is that they are relatively flexible in how they find the widgets to which they have to send an event at replay time: they often can find a component even if its position has changed between runs (e.g. they will click on the right file in a file dialog, even if a new file has changed the layout of the list). However, overall, despite its relative simplicity,

Application	Version	Date	Classes	Description
Arabeske	2.0.1 (stable)	<2004-04	222	Arabeske pattern texture editor
ArgoUML	0.28	2009-03	5349	UML CASE tool
CrosswordSage	0.3.5	2005-10	34	Crossword puzzle editor
Euclide	0.5.2	2009-04	398	Geometry construction kit
FreeMind	0.8.1	2008-01	1909	Mind mapping editor
GanttProject	2.0.9	2008-12	5288	Gantt chart editor
jEdit	2.7pre3	2000-11	1150	Programmer's text editor
JFreeChart Time	1.0.13	2009-04	1667	Chart library, showing temporal data
JHotDraw Draw	7.1	2008-03	1146	Vector graphics editor
Jmol	11.6.21	2009-04	1422	Chemical structure viewer
LAoE	0.6.03	2003-05	688	Audio sample editor
NetBeans Java SE	6.5.1	2009-05	45367	Integrated development environment

Table 4: Applications

Pounder is the most faithful tool with the broadest applicability, even with applications and on a version of Swing developed well after Pounder's terminal release in 2002.

6. DISCUSSION OF PRACTICALITY

The limitations we identified in the five capture & replay tools fall in three different categories:

Incomplete implementation. As our experiments have shown, many existing tools lack support of features used in realistic applications, such as multiple mouse buttons, the use of common dialogs (e.g. to open files), or events on the non-client area of windows (e.g. dragging a window by its frame).

GUI element identification problem. This problem, described by McMaster and Memon [7] in the context of regression testing, even exists when replaying an interaction captured with the same version of the application. Because GUI events are always targeted at specific GUI components, the capture tool needs to store, for each captured event, some information that identifies the target component of that event. At replay time, the replay tool needs to find that component given the information stored by the capture tool. Given that GUI components usually do not have persistent unique identifiers, the capture & replay tools store information such as the component's location, it's class name, it's path in the component tree, or information about the component's other properties (such as the text of a label).

If an application does not behave fully deterministically, or if the environment in which the application runs changes, then a component appearing in one run may not appear, may appear somewhere else, or may appear in a different form, in another run. For example, a game may use a random number generator to compute the computer's next move, a calendaring application may open a calendar on the current date, or a file dialog showing the contents of a folder will show whatever files currently exist.

Moreover, despite Java's platform-independence, replaying a session recorded on a different platform often fails, also because the different implementation of the GUI toolkit (and the different look-and-feel) complicate GUI element identification.

Capture & replay tools partially overcome the GUI element identification problem by using more than one way to identify a component (e.g. by using its path in the component tree, and also storing its class name, and its label). This improves reliability of replay, but comes at the cost of increasing the complexity of capture and replay, and possibly impacting the performance which a performance testing approach is supposed to measure.

Temporal synchronization problem. This problem, which is related to GUI element identification, is a fundamental problem without a simple solution. The issue is that interactive applications, while driven by a user's actions, can also be driven by the system's timer-based actions. For example, a movie player advances to the next movie frame 25 times per second, a clock moves the hand once every second, or a game engine moves a sprite every 50 milliseconds. When a user interacts with an animated component, the user's and the system's actions are synchronized, and together are causing the overall behavior of the application.

However, the capture tools we studied only capture the user's actions. The system's actions are difficult to capture, because they are often not represented as GUI events, and are thus not visible to a GUI capture tool. When the user's actions are replayed, they will not be properly synchronized with the the system's timer-driven actions, and thus will lead to a different overall application behavior. For example, during capture, the user may drag the hand of a clock to adjust time, however, at replay, the clock's hand may be located at a different position, and the drag action thus may not find the component (the hand).

With the increasing use of timer-driven animations in user interfaces, the temporal synchronization problem will grow in importance. In theory, capture & replay tools would have to record all timer-driven system actions in addition to user actions. If developers of interactive applications used a GUI toolkit's standard APIs to perform all their timer-driven activity, capture tools could observe and record all such timer events. GUI toolkits then would need to allow replay tools to filter out application-driven timer events at replay, and to replay the captured timer events instead.

Note that while all of the above limitations are important for GUI performance testing, they also significantly affect functional GUI testing.

7. PERTURBATION

The fact that a tool can successfully record and replay an interactive session does not necessarily mean that performance measurements on a session replayed with that tool accurately reflect the performance of an interactive session with a human user. A capture & replay tool might significantly *perturb* behavior in many ways: (1) the act of recording might perturb the user's or the system's behavior and timing, (2) the fidelity of the recorded interaction log

might be limited, or (3) the act of replaying might perturb the timing. This section presents our results of evaluating the perturbation introduced by the capture & replay tools.

Figure 2 shows the cumulative latency distributions of our interactive sessions (described in Section 5.3) with the twelve applications. For space reasons we only show curves for Windows. The results on the Mac are similar. Each chart in the figures represents one application. The charts have the same structure as the chart explained in Figure 1: the x-axis shows the latency in milliseconds and the y-axis represents the number of episodes that took at least x milliseconds. Each line in a chart represents a tool. We consistently use the same line style for a given tool. If a line for a tool is missing in an application's chart, this means that we did not manage to capture and replay that application's session with that tool. In addition to a line for each tool, each chart also includes a solid line to show the performance measured without any tool. We gathered the data for that line by *manually* repeating the same interactions multiple times. For that curve we neither used a recording nor a replay tool, and thus this represents the unperturbed performance. For each curve (manual or tool-based) we measured the same interaction sequence *five* times. We then computed the average latency distribution over all five runs, which we represent as the curve in the charts, and we show the confidence range (Section 2) for that mean using a hatched pattern.

Focusing on the solid curves representing the unperturbed measurements, Figure 2 shows that almost every application has its own characteristic latency distribution: most of the 12 solid curves have clearly different shapes and locations. For example, Euclide has a smooth curve that bottoms out below 100 ms, Arabeske exhibits a stair-step, but below the 100 ms perceptibility threshold, while Jmol shows a stair-step that extends to 150 ms.

Overall, the curves of the different replay tools look similar to the solid lines. That is, the performance measured *during replay* is relatively close to the performance measured *without tool* ("manual"). However, there often is a statistically significant difference between episode counts of a given latency, i.e. there is no overlap between the hatch patterns of the different curves[7]. We now investigate the most striking differences in more detail.

The Abbot curve on Jmol shows the biggest deviation from the manual latency distribution. Jmol seems to perform much faster when replayed with Abbot than when run manually. The reason for this counter-intuitive result is that Abbot was unable to correctly replay an essential event. It failed to replay a command in a popup menu which reconfigured the visualization to cover the molecule with a difficult to compute surface. Consequently, all the subsequent rendering became much faster, causing the latency distribution to look almost optimal.

The JHotDraw Draw chart shows a significant difference between the manual and the Pounder curves. These interaction sessions contain a large number of mouse drag events (for moving shapes). The shift in the curves can have two reasons: a different *number* of drag events, or a different *latency* of those events. We wrote a small test, and verified that Pounder indeed correctly records and replays the total *number* of mouse drag events. The reason for the shift in the curves is thus not the number of events, but the difference in event latencies. Pounder replays most types of events by directly posting them into the Java event queue. However, Pounder replays mouse drag events using the java.awt.Robot class, which enqueues the events into the underlying native event queue. This approach causes the mouse cursor (rendered by the operating system) to move during drags, but it also adds latency to each event.

The amount of perturbation clearly differs between tools. However, overall we found Pounder to be the tool that most closely matched the original latency distributions.

[7]The confidence intervals are often so tight that the hatch patterns essentially disappear under the average curve.

8. THREATS TO VALIDITY

Our findings are based on the study of five open source capture & replay tools. While the results might differ for commercial GUI testing tools, we believe that some of the described limitations, in particular the temporal synchronization problem, are fundamental problems where possible solutions might significantly perturb performance.

Another threat to validity lies in the small number of Java applications we analyze. However, we believe that the applications we picked represent a realistic sample of the Java rich-client applications used in the field.

The biggest threat probably comes from our choice of interactive sessions. Larger applications like NetBeans are so rich in functionality that any *realistic* interaction covering all features would last many hours, or even days. We thus had to limit our sessions to what we considered the most relevant application features. This problem of picking a representative "input" for an application is a general issue in any quantification of performance.

9. RELATED WORK

The work related to this paper spans two areas: the idea of replaying user interactions, and the general concept of deterministically replaying program executions.

9.1 Replaying User Interactions

Reliably capturing and replaying user interactions is difficult. McMaster and Memon [7] describe the *GUI element identification problem* in the context of GUI test case maintenance and address the problem with a heuristics-based approach. We find that GUI element identification can be a problem even when replaying a recorded interaction on the same version of the application. Moreover, we identify the *temporal synchronization problem*, which is growing in importance with the increasing use of animation in interactive applications.

JRapture [12] is a record/replay tool that attacks the GUI element identification problem with an approach that differs from the tools we studied. It identifies a GUI element by combining the identifier of the thread that created the element with a running count of elements created by that thread. It also differs in that it stores a complete trace of all input and output of the program run, allowing deterministic replay (except for thread scheduling). We believe that storing all data a program reads and writes can significantly perturb performance. JRapture is not available publicly and we thus could not include it in our study.

Grechanik et al. [4] automatically identify which GUI components changed between versions of an application, and then annotate the old version of GUI test scripts with warnings wherever a changed component is used. Their tool turns

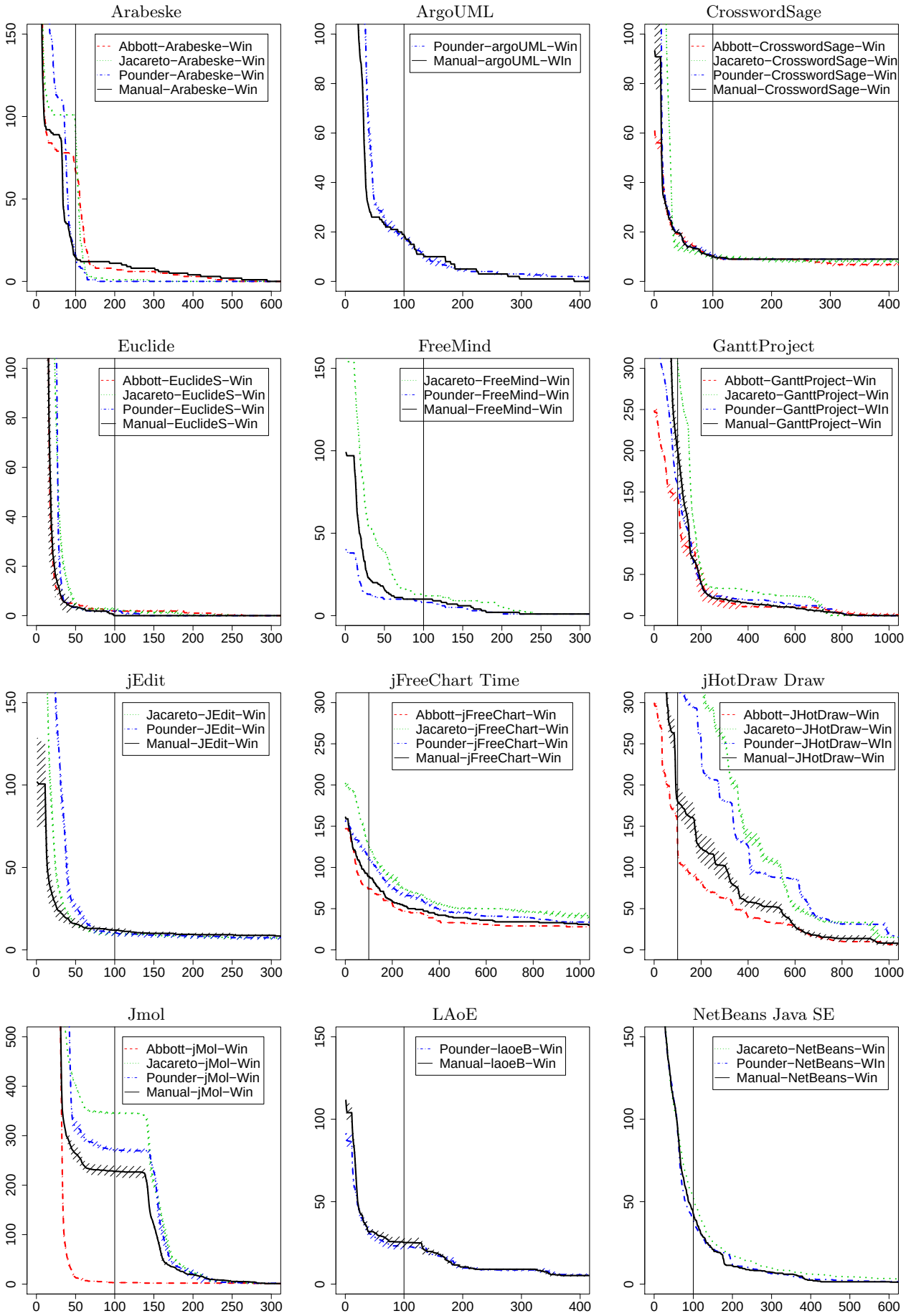

Figure 2: Effect of Tool on Cumulative Latency Distributions

the cumbersome manual evolution of interaction scripts into a semi-automatic approach, thereby greatly reducing the cost of keeping test scripts in synch with evolving applications. Memon [8] describes a similar approach, however, instead of focusing on developer-created test scripts, he focuses on the model-based test scripts generated by his GUITAR infrastructure.

Alsmadi [1] proposes to abstract away from recorded interaction sessions to a higher level of abstraction. Brooks et al. [2] actually implement such an approach. They collect usage profiles, generate a probabilistic model of application usage, and then generate GUI test cases from that model. Such an approaches avoids some of the problems of directly rerunning recorded sessions in different contexts, providing an alternative to repairing recorded sessions.

In this paper we have found the interaction scripts generated by recording tools to be quite brittle. In some situations we could not directly replay a recorded script, even in the same environment and on the same version of the application. We thus believe it would be promising to use the above approaches also for the purpose of repairing interaction scripts for performance testing.

9.2 Low-Level Record/Replay Approaches

Ronsse et al. [11] provide a general description of using record/replay approaches for non-deterministic program executions. Their view is informed by their prior work on RecPlay [10], a record/replay tool for race detection in concurrent programs, where they recorded all synchronization operations between threads. They describe three high-level goals for record/replay approaches: (1) to replay the recorded execution as accurately as possible, (2) to cause as little intrusion as possible, and (3) to operate swiftly. Our evaluation of GUI capture & replay tools for performance comparisons is based on the exact same goals.

Besides their use in race detection, replay approaches have also been used in other domains, such as the live migration of virtual machines [6], post-mortem debugging [9], and intrusion detection [3]. All these approaches operate on a much lower level of abstraction, recording architectural events about program execution. The higher abstraction level of GUI capture & replay approaches leads to less perturbation, and it adds the possibility of replaying a session on a slightly different platform or on a slightly different version of an application.

10. CONCLUSIONS

Today most users interact with computers through applications with graphical user interfaces. Testing the performance of such applications is difficult, because any interactive session necessarily involves users and their non-deterministic behavior.

In this paper we propose an approach for automatically testing the perceptible performance of such applications. We do so by using capture & replay tools. Such tools can record an interactive session with an application and later replay it any number of times. Our approach allows comparative studies of perceptible performance, for example by comparing the application's latency distribution when running on different operating systems or when using different inputs. It also enables automated performance regression testing, the comparison of performance over different versions of an interactive application.

We evaluate different capture & replay tools in terms of their ability to faithfully record and replay interactive sessions, and we find that many such tools are unable to capture realistic interactions with real-world applications. Finally, we study the perturbation of the three most promising tools, by comparing the perceptible performance of automatically replayed application sessions with application sessions driven by real users. We find that the most reliable test automation tool, Pounder, produces performance measurement results that are close to the performance of manually performed interactions.

11. REFERENCES

[1] I. Alsmadi. The utilization of user sessions in testing. In *ICIS '08: Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, pages 581–585, Washington, DC, USA, 2008. IEEE Computer Society.

[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] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71, New York, NY, USA, 2006. ACM.

[4] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[5] 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*, pages 137–146, New York, NY, USA, 2008. ACM.

[6] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110, New York, NY, USA, 2009. ACM.

[7] 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.

[8] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):1–36, 2008.

[9] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[10] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[11] M. Ronsse, K. De Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmüller. Record/replay for nondeterministic program executions. *Commun. ACM*, 46(9):62–67, 2003.

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