Software Qual J (2011) 19:801–839
DOI 10.1007/s11219-011-9135-x

Automated GUI performance testing

Andrea Adamoli · Dmitrijs Zaparanuks · Milan Jovic · Matthias Hauswirth

Published online: 3 April 2011

Abstract A significant body of prior work has devised approaches for automating the functional testing of interactive applications. However, little 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 and replay tools for GUI performance test automation. Besides confirming the severity of the previously known *GUI element identification problem*, we also describe 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. Based on an investigation of Pounder's approach, we further improve its robustness and reduce its perturbation. Finally, we demonstrate in a set of case studies that the conclusions about perceptible performance drawn from manual tests still hold when using automated tests driven by Pounder. Besides the significance of our findings to GUI performance testing, the results are also relevant to capture and replay-based functional GUI test automation approaches.

This article is an extended version of our AST 2010 paper Jovic et al. (2010).

A. Adamoli (✉) · D. Zaparanuks · M. Jovic · M. Hauswirth
Via Giuseppe Buffi 13, 6904 Lugano, Switzerland
e-mail: andrea.adamoli@usi.ch

D. Zaparanuks
e-mail: dmitrijs.zaparanuks@usi.ch

M. Jovic
e-mail: milan.jovic@usi.ch

M. Hauswirth
e-mail: matthias.hauswirth@usi.ch

Keywords Performance testing · Graphical user interfaces · Test automation · Perfomance analysis

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 measuring the distribution of system response times to user requests (Jovic and Matthias Hauswirth 2008). 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. Capture and 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 and replay approach that incurs little overhead while still being able to replay realistically complex sessions.

In this paper, we evaluate capture and replay approaches as implemented in a set of open-source Java GUI testing tools. 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 provides the background on the structure and behavior of interactive applications, in particular on GUI applications written in Java. Section 3 surveys prior work on GUI testing. Section 4 introduces GUI capture and replay tools. Section 5 presents our methodology for evaluating the fitness of such tools for recording and replaying long, realistic event sequences. Section 6 applies that methodology to evaluate five existing capture and replay tools. Section 7 discusses our findings. Section 8 investigates the limitations of Pounder, the tool with the best evaluation results, and it discusses how we improved Pounder as a result of our study. Section 9 provides five case studies to highlight use cases of GUI performance test automation. Section 10 discusses threats to the validity of our study, Sect. 11 summarizes related work, and Sect. 12 concludes.

2 Interactive applications

Interactive applications are event-based systems: they wait for and process a never-ending sequence of user requests. Java's AWT/Swing, like most GUI toolkits, represents user requests as "events". In Java, an event is a normal Java object. Event classes are subtypes of `java.util.EventObject`. Each event describes an action that applies to a specific object. In most cases, this object affected by the event is a GUI component (e.g., a `MouseEvent` may represent a mouse click on a `JButton` component).

While a GUI application is running, the toolkit creates an event object for each user interaction with the keyboard or mouse (1 in Fig. 1), and it enqueues that event object into an event queue. The event dispatch thread, the one thread in Java GUI applications that executes all GUI-related code, dequeues one event object after the other, and dispatches it. Dispatching an event object often means sending it (2) to the corresponding GUI component (an object of a subtype of `java.awt.Component`).

Unfortunately, this simple model has many exceptions and boundary cases.

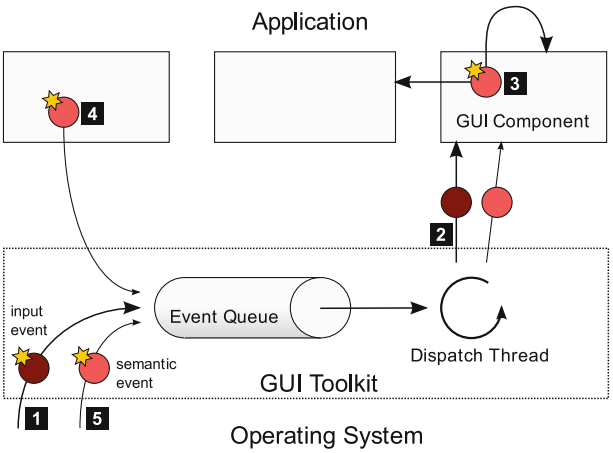

Fig. 1 Event dispatch model

First, not all events are actually dispatched to components. For example, the dispatcher handles events of type `java.awt.ActiveEvent` by invoking a dispatch method on the event itself. Those events are not related to a GUI component at all. They represent arbitrary actions that are to be executed in the GUI thread. Often they are used by background threads to cause the GUI thread to update the state of the GUI on their behalf.

Second, not all events are enqueued in the event queue; so-called semantic events may originate in the application itself (3 and 4), often as a result (3) of low-level events. For example, a user may press and release the mouse button (thereby causing two low-level `MouseEvents`, one for the pressing and one for the releasing of the button) while the mouse cursor is located over a GUI component that represents a text field. The text field, listening to `MouseEvents`, may then create a `CaretEvent` and send it to its listeners. The `CaretEvent` is a semantic event that represents the fact that the caret (text cursor) has been moved. Semantic events like the `CaretEvent` in this example are usually not sent through the event queue. Instead, they are directly sent to all `CaretListeners` registered with the text field by invoking their `caretUpdate(CaretEvent)` methods.

Third, some parts of the GUI visible to the user are not reflected as objects in the Java program. For example, the window decoration, including the title bar and the close or maximize buttons, is directly drawn by the native window system. Mouse activities over these areas are not sent to the Java application. Instead, the application only receives certain semantic events (5), such as a `WindowEvent` when the user clicked the close button.

3 Survey of automated GUI testing approaches

Figure 2 provides a high-level characterization of automated GUI testing approaches. In this section, we use this characterization to classify a body of 50 research papers published on this topic. We started our survey with the top-ranked papers by relevance in searches for "GUI testing" and "GUI capture replay" Memon (2008), Memon et al. (2005), Li et al. (2007), Xie and Memon (2008), Nguyen et al. (2010), Yuan and Memon (2010), Brooks and Memon (2007), Sun and Jones (2004), El Ariss et al. (2010) in the ACM digital library. Given that most of this work was focusing on desktop applications, we added further papers related to GUI testing of web applications. We then performed the transitive closure over the relevant papers each paper cites.

3.1 Characterization of GUI testing automation

The graph in Fig. 2 outlines the GUI testing process. It consists of two kinds of nodes: activities (elliptical) and data collections (rectangular), with each edge connecting an activity and a data collection. A key goal of automated GUI testing is to eliminate the need for a human user during testing, that is, to *automatically* run an interactive application (the **play** activity), performing a given sequence of user interactions (**event sequence** data collection). The surveyed approaches differ in which of the nodes of our figure they address.

The common capture and replay approach to GUI testing includes the **record** process, which records an event sequence while a user interacts with the application. However, many GUI testing approaches use **models** to abstract away from concrete event sequences. A model represents a set of possible event sequences. Given a model, approaches can **instantiate** concrete event sequences, which they can then play. The model can consist of

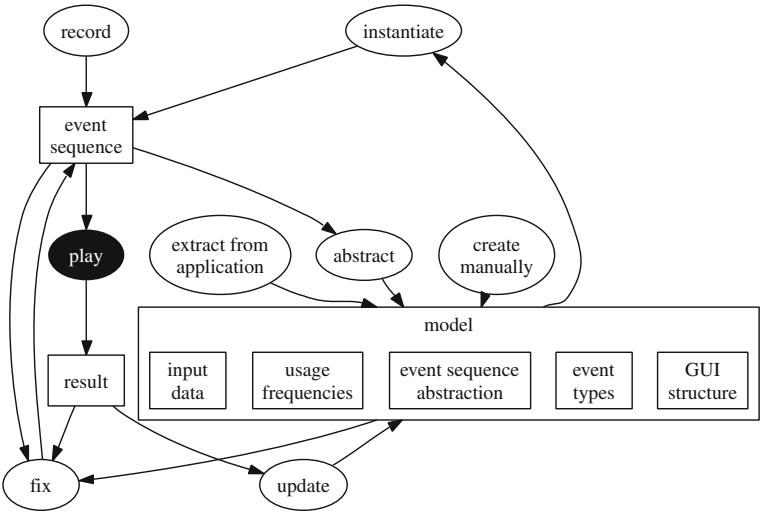

Fig. 2 Characterization of GUI testing approaches

an **event sequence abstraction**, such as an event flow graph, state machine, or Markov model. The model may include information about possible **event types**. It can involve information about the **GUI structure**, which is usually represented as a forest where each top-level window contains a tree of components. The model also may involve **usage frequencies**, representing information about which components or events are most important in practice. Finally, models may separate **input data** (such as the set of strings that could be entered into text fields) from the event sequence abstraction.

A model can be created and maintained in different ways. An approach may **abstract** a model from one or more concrete event sequences. Alternatively, a model may be **created manually**, maybe even before the application exists. A model, or parts of a model (such as the GUI structure), may be **extracted from the application** using static or dynamic analysis. Finally, feedback-driven approaches **update** the model based on **results** of playing back event sequences. The results may correspond to test failures, to crashes at play time, or to outcomes of dynamic analyses performed during play time.

The results are not only used in model-based approaches. Even traditional capture and replay approaches may use the results of running a sequence in order to **fix** that sequence. Moreover, fixing a concrete event sequence may also use information from a model, for example information about changes in a GUI structure between application versions.

3.2 Survey of prior work on GUI test automation

Table 1 classifies the 50 surveyed papers according to the above characterization. It contains one column for each node in Fig. 2. The columns related to models are grouped in the middle. The top part of the table consists of 18 papers that include the *record* activity, while the 32 papers in the bottom part do not involve the recording of concrete event sequences.

For the papers involving recording, the top 6 papers operate on concrete event sequences and thus represent traditional capture and replay approaches, while the bottom 12 papers involve abstract models of user interactions. The last two of those 12 papers also

Table 1 Survey of GUI testing approaches

Paper	Model														
	Record	Event sequence	Play	Result	Update	Abstracted event sequence	GUI structure	Input data	Usage frequencies	Event types	Create manually	Extract from application	Abstract	Instantiate	Fix
Steven et al. (2000)	X	X	X												
Lowell and Stell-Smith (2003)	X	X	X												
Meszaros (2003)	X	X	X												
Li and Mengqi Wu (2004)	X	X	X												
Mitchell and Power (2004)	X	X	X												
Ruiz and Price (2007)	X	X	X												
Liu et al. (2000a)	X	X	X				X						X	X	
Liu et al. (2000b)	X	X	X				X						X	X	
Memon et al. (2001)	X	X	X			X	X			X		X	X	X	
Elbaum et al. (2003)	X	X	X			X							X	X	
Sampath (2004)	X	X	X						X	X			X	X	
Elbaum et al. (2005)	X	X	X			X							X	X	X
Brooks and Memon (2007)	X	X	X			X			X				X	X	
Alsmadi (2008)	X	X	X			X			X				X	X	
Deursen and Mesbah (2010)	X	X	X			X							X	X	
El Ariss et al. (2010)	X	X	X				X			X	X	X		X	X
Kasik and George (1996)	X	X	X	X	X	X							X	X	X
Memon (2008)	X	X	X	X	X	X	X				X		X		X
Shehady and Siewiorek (1997)		X	X				X	X		X	X			X	
Yang et al. (1999)		X	X			X	X	X			X	X		X	

Table 1 continued

Paper	Model														
	Record	Event sequence	Play	Result	Update	Abstracted event sequence	GUI structure	Input data	Usage frequencies	Event types	Create manually	Extract from application	Abstract	Instantiate	Fix
White and Almezen (2000)		X	X			X	X					X		X	
Belli (2001)		X				X					X			X	
Ricca and Paolo Tonella (2001)		X	X			X	X	X			X	X		X	
Lucca et al. (2002)		X	X				X					X		X	
Sun and Jones (2004)		X	X			X					X			X	
Memon and Qing Xie (2005)		X	X			X	X				X	X		X	
Memon et al. (2005)		X	X			X	X				X	X		X	
Xie (2006)		X	X			X	X					X		X	
Grechanik et al. (2009)		X	X				X					X			X
Li et al. (2007)		X	X			X	X				X			X	
Marchetto et al. (2008a)		X	X				X					X		X	
Marchetto et al. (2008b)		X	X				X					X		X	
Hackner and Memon (2008)		X	X			X						X		X	
Strecker and Memon (2008)		X	X				X				X			X	
Brooks et al. (2009)		X	X			X					X			X	
Chinnapongse et al. (2009)		X	X			X					X			X	
McMaster and Memon (2009)		X	X			X	X				X	X		X	
Mesbah and van Deursen (2009)		X	X				X					X		X	

Table 1 continued

Paper	Model														
	Record	Event sequence	Play	Result	Update	Abstracted event sequence	GUI structure	Input data	Usage frequencies	Event types	Create manually	Extract from application	Abstract	Instantiate	Fix
Mu et al. (2009)		X	X			X	X	X			X			X	
Silva et al. (2009)		X	X				X					X		X	
Chang et al. (2010)		X	X			X									
Nguyen et al. (2010)		X	X			X	X					X		X	
Lindvall et al. (2007)		X	X	X	X	X			X		X	X		X	
Xie and Memon (2007)		X	X	X	X	X						X		X	
Yuan and Memon (2007)		X	X	X	X	X						X		X	
McMaster and Memon (2008)		X	X	X	X	X			X			X		X	
Ruiz and Price (2008)		X	X	X	X	X	X				X			X	
Xie and Memon (2008)		X	X	X	X		X				X			X	
Yuan et al. (2009)		X	X	X	X	X						X		X	
Yuan and Memon (2010)		X	X	X	X	X	X					X		X	

include feedback, which means that the model or the sequence is updated based on results of prior runs.

All the 32 papers that do not involve any recording represent model-based testing approaches. Of those, the last 8 papers update their models based on feedback from prior runs.

3.3 Fitness for GUI performance testing

To find performance problems in real applications, the *length of the event sequences* played during testing is important. Sequences representing only one or two events are often used for functional testing. They represent a form of unit test. Slightly longer sequences could be considered integration tests, as they often cover some interactions between components. To find performance problems, however, event sequences need to be significantly longer, so that the underlying system can reach the steady-state behavior that is normal in real-world usage. In our survey, we found that more than 50% (28) of the papers do not deal with long sequences, thus their tests are not very suitable for performance testing.

A second problem of using GUI testing tools for performance testing is their use of harnesses and mock objects. Those artifacts represent deviations from the real-world setup and thus can affect the observed performance.

Out of the 50 related papers, 44 represent model-based approaches. The use of models implies the automatic generation of event sequences (instantiation). Thus, model-based approaches allow the generation of an arbitrary number of sequences of arbitrary lengths. However, the surveyed model-based approaches did not primarily focus on producing long and realistic sequences. Their main goal is to increase test coverage, according to various coverage criteria. Many approaches aim at achieving high coverage while keeping sequences as short as possible, especially given that exploring the model using longer sequences can lead to a combinatorial explosion of the number of possible sequences. Five approaches include usage frequencies in their models. While such information could be used to generate more realistic event sequences, this is not the focus of those papers.

Current work on GUI test automation does not explicitly focus on exercising applications using realistic, long event sequences. As a consequence, evaluations of that work focus on other aspects, such as the reduction in testing time and the improvement of coverage. In this paper, we fix this gap: we evaluate GUI test automation approaches, in particular the more mature category of capture and replay tools, for their ability to handle real-world usage sessions.

4 Capture and replay tools

GUI capture and replay tools have been developed as a mechanism for testing the correctness of interactive applications with graphical user interfaces. Using a capture and 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 and replay tools thus support fully automatic regression testing of graphical user interfaces.

4.1 Capturing interactions

The complexities inherent in interactive Java applications described in Sect. 2 make it difficult to build a tool that can accurately capture all user interactions. A naive tool may capture only low-level mouse and keyboard events, considering low-level events the root causes of all behavior of the application (because semantic events are usually created as a result of some low-level event). Such a tool will fail to capture user activity in parts outside the application's control (such as the window's title bar). Moreover, mouse events that include the coordinates of the mouse pointer are very fragile. The coordinates in a mouse event are required to determine to which GUI component the `MouseEvent` needs to be sent. If the layout of the GUI changes slightly (e.g., in a new version of the program or on a platform with a different look-and-feel), the x and y coordinates may now represent a location over a different component. For this reason, recording high-level semantic events would be preferable over the use of low-level events.

Unfortunately, there is no complete set of semantic events. Applications may add their own event classes (and they frequently do). Moreover, components may handle some mouse events directly, without ever sending out higher level semantic events. Thus, capture tools focusing exclusively on semantic events essentially will miss potentially relevant user interactions.

4.2 Persisting interactions

Once a tool builder has decided how to capture user interactions, she has to decide how to represent these interactions in a persistent form. A capture and replay tool has to persist events between different executions of the application and virtual machine. The capture tool serializes events to a log file, and the replay tool deserializes them from that file. Thus, there is a question about how to represent events in persistent form.

Current tools generally use an XML-based file format to store events. Thus, the tools map Java event objects to XML elements and the fields of the Java objects to attributes of the XML elements. Most of the attributes of events are simple scalar values, such as the x and y coordinates in a `MouseEvent`, the key code of the key pressed in a `KeyEvent`, or the event's type (often available as an int field in the event object). Unfortunately, one of the most important attributes of events is an object reference: the target component to which the event is to be dispatched. Serializing and deserializing that attribute is what causes many of the problems in existing capture and replay tools (this is called the "GUI Element Identification Problem" McMaster and Memon (2009)).

5 Evaluation methodology

In this section, we present a methodology[1] for evaluating the fitness of GUI capture and replay tools for automated GUI performance testing. While we focus on the tools' fitness for performance testing, the first two parts of our methodology are concerned with the tools' fitness for functional testing and are thus of more general use.

Approach. We first evaluate the tools' *functional accuracy using small test applications*. Our test applications are minimal GUI applications, each just exhibiting one specific GUI

[1] The artifacts used in this methodology are available at http://www.sape.inf.usi.ch/pounder and will be submitted to the *Community Event-based Testing* collection at http://www.comet.unl.edu/.

feature. We use the tools to record a short user session on each application, and we check that replaying that session leads to identical behavior. This part of the evaluation essentially corresponds to *unit testing* the tools.

Second, we evaluate the tools' *functional accuracy using real-world applications*. Being able to record and replay sessions on realistic applications is harder: the tools need to deal with the many intricacies of large programs, with complicated user interfaces and with unconventional uses of GUI features, and they need to be able to record and replay interactive sessions that are much longer than the few events occurring in the test applications. This second part of the evaluation corresponds to *system testing* the tools.

Finally, we evaluate the *accuracy of performance measurements* under the tool. This is important because the use of the capture and replay tool affects the overall behavior of the system, and this may significantly perturb the performance measurement. This last part of the evaluation performs the same tasks on the same applications; however, it evaluates performance accuracy instead of functional accuracy.

Measures. In all three of the above parts, we need some measure of fitness, or accuracy, of the tool. For the two functional evaluations, this measure corresponds to an oracle that can tell whether the behavior of the application during replay was identical to the behavior during recording. Primarily, we act as oracles ourselves, observing the application during capture, and checking for identical behavior during replay. However, we also use automated oracles: to determine the equivalence of capture and replay behavior, we profile both executions and compare the profiles. We use two kinds of profiles. First, we count the number of events of each distinct event type (e.g., mouse press, key release). Second, we count the number of executions for each listener in the program (e.g., FileOpenAction or InsertCharacterAtKeyPressListener).

For the performance evaluation, we measure perceptible performance in the form of the cumulative latency distribution. We measure the latency of each event, and we compare the distribution during replay to the latency distribution during manual operation.

5.1 Accuracy on test applications

Table 2 shows the nine 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 and replay tool can faithfully maintain timing

Table 2 Test applications

Application	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

information. It provides a component that toggles between two colors, red and white, every 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).

5.2 Accuracy on real-world applications

Table 3 shows the twelve real-world 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 in experiments in prior work on GUI testing (Brooks and Memon 2007).

5.2.1 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.

Table 3 Real-world applications

Application	Version	Date	Classes	Description
Arabeske	2.0.1 (stable)	<2004–04	222	Arabeske pattern editor
ArgoUML	0.28	2009–03	5,349	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	1,909	Mind mapping editor
GanttProject	2.0.9	2008–12	5,288	Gantt chart editor
jEdit	2.7pre3	2000–11	1,150	Programmer's text editor
JFreeChart Time	1.0.13	2009–04	1,667	Chart library, temporal data
JHotDraw Draw	7.1	2008–03	1,146	Vector graphics editor
Jmol	11.6.21	2009–04	1,422	Chemical structure viewer
LAoE	0.6.03	2003–05	688	Audio sample editor
NetBeans Java SE	6.5.1	2009–05	45,367	IDE

Table 4 Application sequences

Application	Number of sequences	Sequence length	Testing time [min]	Users involved
Arabeske	4	1,560	5	2
ArgoUML	6	1,834	7	3
CrosswordSage	6	2,483	3	3
Euclide	6	1,632	3	3
FreeMind	4	2,161	5	2
GanttProject	6	1,346	6	3
jEdit	6	2,329	7	3
JFreeChart Time	6	1,658	5	3
JHotDraw Draw	3	2,364	9	3
Jmol	6	2,364	7	3
LAoE	6	1,821	5	3
NetBeans Java SE	4	3,421	7	2

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 modified our environment or adjusted the interaction slightly and went back to step 2.

5.2.2 Interactive sessions

We devised a realistic interactive session for each application and recorded it with each of the capture tools. We had to do 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, based on the results from running the small test applications, 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.

Table 4 shows our experimental setup for the chosen real-world applications. The event sequences correspond to realistic user sessions. For each application, we recorded multiple sequences, varying the size of the documents we edited in the application. With larger document sizes, we were stressing the application more, which, in many cases, reflected on the performance of the given application. This fact leads to increased execution time in both manual and replayed sessions. The duration of sessions is reflected in two columns: "Testing Time" (average, in minutes) and "Sequence Length" (average number of events).

5.3 Performance 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

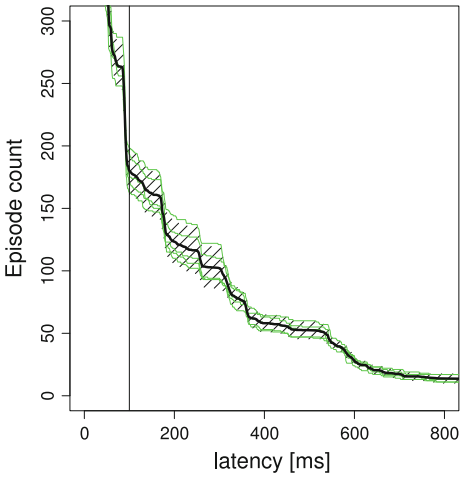

Fig. 3 Cumulative latency distribution for characterizing perceptible performance

accurately reflect the performance of an interactive session with a human user. A capture and 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.

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 (Jovic and Matthias Hauswirth 2008)) 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 3 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 3 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.

6 Evaluation

In this section, we follow the evaluation methodology we introduced in Sect. 5 to evaluate the five capture and replay tools in Table 5. All these tools are written in pure Java and are

Table 5 Open-source java capture and replay tools evaluated in this study

Tool	Version	Date	Status	Classes
Abbot	1.0.2	2008–08	Active	577
Jacareto	0.7.12	2007–03	(Active)	1,085
Pounder	0.95	2007–03	Dead	156
Marathon	2.0b4	2009–01	Active	694
JFCUnit	2.08	2009–01	Dead	150

available as open-source. They are all capable of recording and replaying interactive sessions of applications based on Java's standard Swing GUI toolkit.

Abbot[2] 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[3] is a GUI capture and replay tool supporting the creation of animated demonstrations, the analysis of user 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 capture and replay tool we use in this paper.

Pounder[4] 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 with 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 5).

Marathon[5] 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[6] is an extension that enables GUI testing based on the JUnit[7] 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.

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

[2] http://www.abbot.sourceforge.net/.

[3] http://www.jacareto.sourceforge.net/.

[4] http://www.pounder.sourceforge.net/.

[5] http://www.marathontesting.com/.

[6] http://www.jfcunit.sourceforge.net/.

[7] http://www.junit.sourceforge.net/.

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.

6.1 Accuracy on test applications

Table 6 shows the results we obtained by running the five capture and 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.

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.

Scrolling. Pounder is the only tool that supports auto-scrolling (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.

Table 6 Test results

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

ComboBox. Abbot has problems with heavy-weight popup windows and was unable to replay interactions with drop-down 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 evaluation on Abbot, Jacareto, and Pounder.

6.2 Accuracy on real-world application

We now show which tools were able to properly capture and replay our interactions with real-world Java applications. Table 7 presents one column 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 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.

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

Table 7 Tool applicability

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			✓	✓	✓	✓

6.3 Performance perturbation

This section presents our results of evaluating the degree to which capture and replay tools perturb the measurement of perceptible performance.

Figures 4 and 5 shows the cumulative latency distributions of our interactive sessions (described in Sect. 5.2.2) with the twelve applications running on 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 Fig. 3: 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 for that mean using a hatched pattern.

Focusing on the solid curves representing the unperturbed measurements, Figs. 4 and 5 show 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[8]. 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 that 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

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

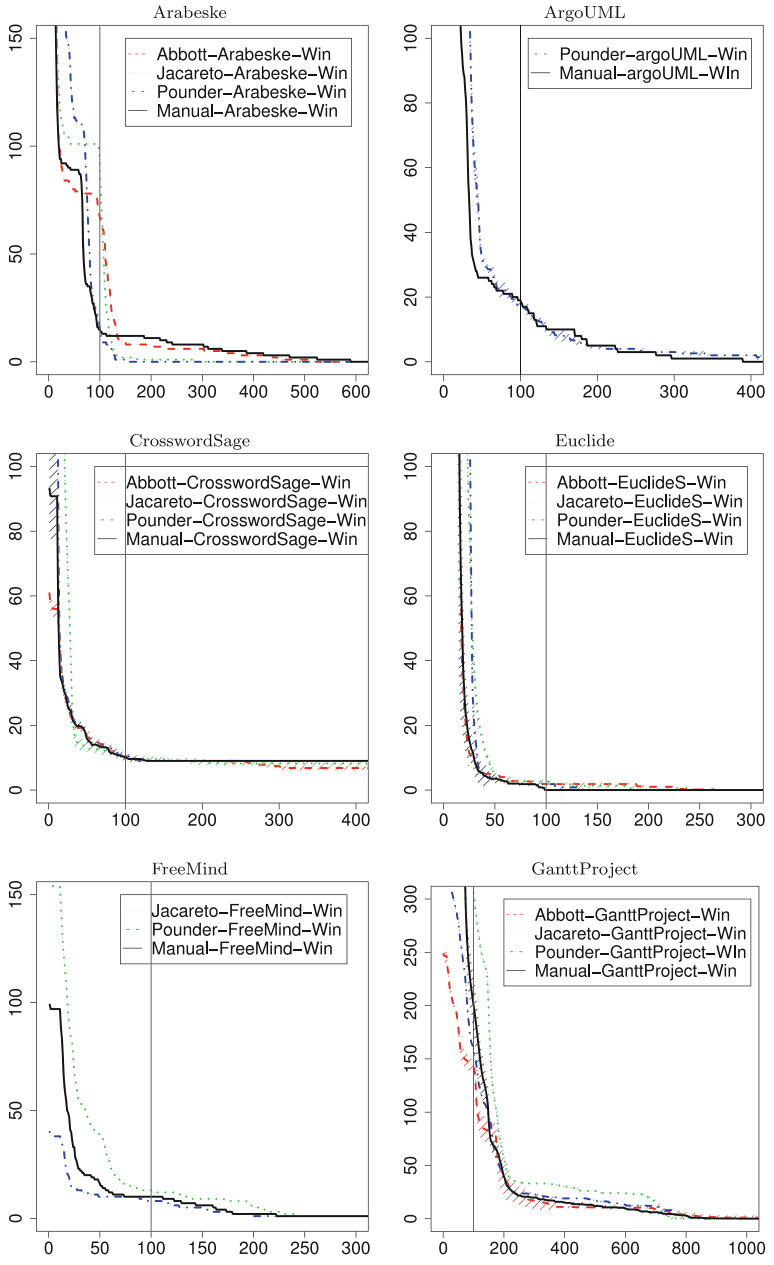

Fig. 4 Effect of tool on cumulative latency distributions (Part 1/2)

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.

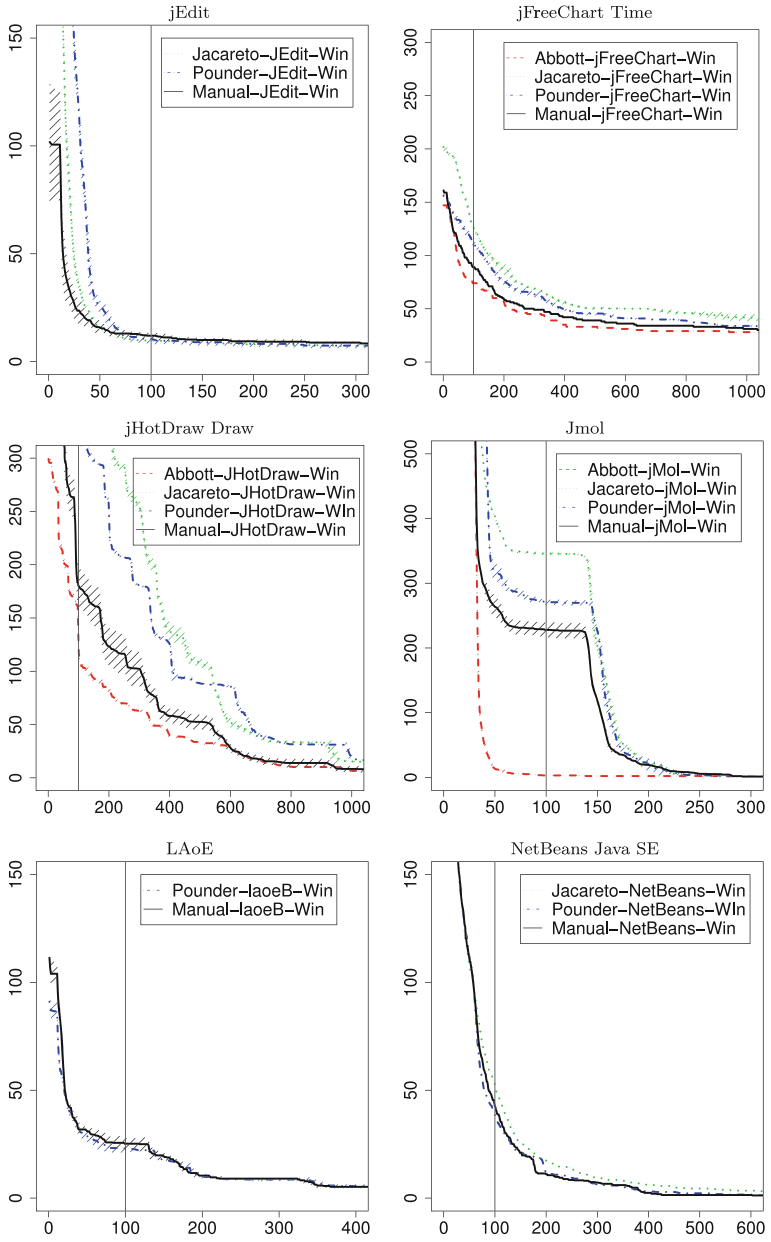

Fig. 5 Effect of tool on cumulative latency distributions (Part 2/2)

7 Discussion

The limitations we identified in the five capture and replay tools fall into three different categories. Note that while all of these limitations are important for GUI performance testing, they also significantly affect functional GUI testing.

7.1 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). This kind of limitation is not fundamental and can be overcome by implementing the missing support.

However, the limitation is a direct consequence of the complexities involved in capturing and replaying GUI events in Java. Java provides its own GUI toolkit (AWT/Swing) that implements an abstraction layer that hides the underlying native GUI toolkit of the operating system. While this provides the advantage of being able to write platform-independent GUI applications, it significantly increases the complexity of properly capturing and replaying GUI events.

7.2 GUI element identification problem

This problem, first identified by McMaster and Memon (McMaster and Memon 2009) 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 and 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 and replay tools partially overcome the GUI element identification problem using more than one way to identify a component (e.g., 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.

7.3 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 ms. 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, a capture tool only captures 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 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 and replay tools would have to record all timer-driven system actions in addition to user actions. This can only happen, if developers of interactive applications use a GUI toolkit's standard APIs to perform their timer-driven activity, thereby allowing capture tools to observe all such timer events. Alternatively, the capture and replay tools could capture all inputs (such as mouse, keyboard, timer, file, and network activity) on a low level (Steven et al. 2000). However, such an approach would risk to significantly perturb the performance we want to measure.

8 Improving Pounder

In the prior sections, we have shown that Pounder is the most appropriate of the existing open-source capture and replay tools for Java. In this section, we study Pounder's implementation, and we describe improvements to overcome some of its limitations. We implemented the proposed improvements and integrated them into the official Pounder project[9].

8.1 Pounder's capture and replay approach

As outlined in Sect. 7.1, GUI event capture and replay is particularly challenging in Java because of the involvement of two GUI toolkits: the Java AWT and the toolkit of the underlying operating system. The Java AWT is not a complete GUI toolkit, it just provides an abstraction layer over the underlying OS GUI toolkit. Both toolkits know the concept of events and each toolkit maintains its own event queue. Events that originate from the user (such as mouse or keyboard actions), flow through the OS event queue, are forwarded to the Java event queue and finally are handled by the Java application.

One difficulty in the implementation of Java-based capture and replay tools is that they cannot replay all events *at the same level* at which they capture them. To *record* events, a tool has to hook into Java AWT's event queue to track all events dispatched through that queue. To *replay* the events, the tool sometimes has to post them to the OS event queue. This second requirement is due to the fact that the OS *also* reacts to certain kinds of events before forwarding them to Java. If the replay tool just replayed them through the Java event queue, the OS would not know about them and would not update its state.

As a consequence, Java capture and replay tools need to use three different ways to replay events, depending on the kind of event: (1) by posting an event into the Java event

[9] Information about and links to our contributions to Pounder are available at http://www.sape.inf.usi.ch/pounder.

queue, (1) by posting an event into the operating system event queue, or (3) by invoking AWT methods that call into the OS and ultimately cause an event to be posted.

Posting to the *Java* event queue works via `EventQueue.postEvent(event)`, where `event` can be any `AWTEvent` subtype (such as `MouseEvent`). Posting to the *OS* event queue is supported via Java AWT's `Robot` class. That class only understands a small subset of events. Table 8 shows the corresponding ways for posting events to the Java event queue versus the OS event queue. It lists all ways supported by `Robot`. Those are related to either mouse or keyboard input. `Robot` does not support MOUSE_CLICKED or KEY_TYPED, because those "synthetic" events are indirectly triggered by other events. Moreover, it does not support MOUSE_DRAGGED, because that event can be generated with a `Robot.mouseMove()*` while a mouse button is pressed.

One example for the difference of posting to the OS versus the Java event queue is the posting of mouse motion events. Posting a mouse motion event to the OS event queue causes the mouse pointer (rendered by the OS, not by Java) to actually move on screen. Posting the event to the Java event queue does not move the mouse pointer and only tells the *application* about the movement. As this example shows, behaviors related to resources under control of the OS (e.g., repositioning the screen representation of the mouse pointer, or focusing a window) cannot be replayed just by posting to the Java event queue.

Unfortunately, the `Robot` class does not provide mechanisms for all possible kinds of OS-relevant events. For this reason, a replay tool sometimes needs to call Java API methods, instead of posting events, to tell the OS to perform a certain behavior. For example, to cause a window to receive the focus, the WINDOW_GAINED_FOCUS event observed at recording time is just a notification to the application that the window received the focus (a state change in the operating system's GUI toolkit). Posting that event to the Java event queue would not cause the window to receive the focus (it would just make the application believe that the window received the focus). Thus, the replay tool has to use other Java AWT methods, such as `Window.requestFocus()`, to affect the state of OS resources and cause events to be posted.

Pounder thus has to use all three approaches for replaying events. It favors the lighter-weight Java event queue and only falls back on the `Robot` or other Java API methods in certain situations. Table 9 lists all the events captured by Pounder, ordered by replay approach (Java event queue, OS event queue, Java API method calls).

In general, Pounder only captures "user-generated" events. It does not capture "synthetic" events that are generated by software. For example, it does not capture the synthetic

Table 8 Input event mapping between OS and java

Java AWT event	Robot method to trigger OS event
MouseEvent.MOUSE_PRESSED	Robot.mousePress()
MouseEvent.MOUSE_RELEASED	Robot.mouseRelease()
MouseEvent.MOUSE_CLICKED	(None)
MouseEvent.MOUSE_MOVED	Robot.mouseMove()
MouseEvent.MOUSE_DRAGGED	Robot.mouseMove()*
MouseEvent.MOUSE_WHEEL	Robot.mouseWheel()
KeyEvent.KEY_PRESSED	Robot.keyPress()
KeyEvent.KEY_RELEASED	Robot.keyReleased()
KeyEvent.KEY_TYPED	(None)

event representing a move of a non-Window component, because that movement is caused by some other event (e.g., a mouse drag or the movement of the containing window). Correctly determining the set of user-generated events is crucial for the correct functioning of the tool. If the tool accidentally captures a synthetic event, that event will be replayed twice: once by Pounder and once (correctly) by the system as a result of the replay of its triggering event.

MOUSE_CLICKED in Table 9 is a synthetic event, and thus, Pounder neither captures nor replays it (we only list it in this table to provide a complete list of all kinds of mouse events). When the user presses down a mouse button, she creates a MOUSE_PRESSED event. When she later releases the mouse button, she creates a MOUSE_RELEASED event. Both of these events are user generated. The GUI toolkit, as a consequence of observing a MOUSE_PRESSED followed by a MOUSE_RELEASED without any intervening MOUSE_MOVED event or time delays, will then generate a synthetic MOUSE_CLICKED event. Thus, the user-generated MOUSE_RELEASED event may trigger a synthetic MOUSE_CLICKED. Thus, `Robot.mousePress()` followed by `Robot.mouseRelease()`, without any interleaved significant moves or time delays, will trigger the GUI toolkit to generate a MOUSE_CLICKED event.

8.2 Improving robustness

While Pounder is the most robust tool we studied, using Pounder to capture and replay realistically long sessions still often leads to errors during replay that require the re-recording or manual editing of the trace. In this subsection, we study how to improve Pounder's robustness.

8.2.1 Retry of events that fail to play back

Pounder has a mechanism to repeatedly retry playing back an event if it fails. Failed playbacks are related to the *component identification* and the *temporal synchronization* problem: if Pounder cannot identify the target component of a given event, it will sleep briefly and try again, until it can find the component or until it runs out of retries. Thus, to study the effect of Pounder's retry approach, we instrumented Pounder to count the number of retries.

In our experiments, we found that if a playback failed then its retries never succeeded. Either Pounder did not have to retry and succeeded in playing back the event at the first try or it retried but failed in all of the retries.

While we did not implement this idea, one possibility to further improve Pounder's (and any GUI replay tool's) robustness could be to retry to play back not just the currently failing event, but also the previous N events before the failing event. Thus, if a retry failed because some components were not set up correctly, replaying some of the preceding events might correct that incomplete or incorrect setup (it might, however, also make things worse).

8.2.2 Temporal synchronization

Studying the recorded Pounder sessions that failed to replay, we found an interesting recurring problem: some mouse events, when replayed, were directed at components that were no longer visible. The reason for this was that a preceding event was causing the

Table 9 Pounder's capture and replay approach

Pounder item	Recorded java event	Replay approach
InputMethodItem	InputMethodEvent. INPUT_METHOD_TEXT_CHANGED	EventQueue.postEvent(new InputMethodEvent());
InputMethodItem	InputMethodEvent. CARET_POSITION_CHANGED	EventQueue.postEvent(new InputMethodEvent());
KeyItem	KeyEvent. KEY_PRESSED	EventQueue.postEvent(new KeyEvent());
KeyItem	KeyEvent. KEY_RELEASED	EventQueue.postEvent(new KeyEvent());
KeyItem	KeyEvent. KEY_TYPED	EventQueue.postEvent(new KeyEvent());
MouseWheelItem	MouseWheelEvent. MOUSE_WHEEL	EventQueue.postEvent(new MouseWheelEvent());
MouseMotionItem	MouseEvent. MOUSE_ENTERED	EventQueue.postEvent(new MouseEvent());
MouseMotionItem	MouseEvent. MOUSE_EXITED	EventQueue.postEvent(new MouseEvent());
MouseMotionItem	MouseEvent. MOUSE_MOVED	EventQueue.postEvent(new MouseEvent());
MouseMotionItem	MouseEvent. MOUSE_DRAGGED	Robot.mouseMove();
MouseClickItem	MouseEvent. MOUSE_PRESSED	Robot.mouseMove(); Robot.mousePress();
MouseClickItem	MouseEvent. MOUSE_RELEASED	Robot.mouseMove(); Robot.mouseRelease();
MouseClickItem	MouseEvent. MOUSE_CLICKED	nothing
WindowGainedFocusItem	WindowEvent. WINDOW_GAINED_FOCUS	Window.requestFocus();
WindowStateChangedItem	WindowEvent. WINDOW_STATE_CHANGED	Frame.setExtendedState();
WindowMovedItem	ComponentEvent. COMPONENT_MOVED	Window.setLocation(); Window.repaint();
WindowSizeChangedItem	ComponentEvent. COMPONENT_RESIZED	Window.setSize(); Window.validate();

component to become hidden; however, due to the asynchronous nature of some native window system operations, the component did not get hidden immediately, but stayed visible long enough for some additional mouse events to be recorded on it. If, at replay time, the component was hidden more quickly, then Pounder could not associate the subsequent mouse events with that component anymore.

After identifying that problem, we added a work-around to Pounder, so that it checks whether the component is visible and if not makes it visible before playing back the event.

This fix allowed us to play back many sessions that we previously were unable to replay without manually fixing them.

8.2.3 Component identification

Another prevalent problem in Pounder was related with component identification. We noticed that Pounder often wrongly replayed mouse clicks on some buttons of some standard dialogs: it replayed the clicks, but it clicked the wrong button. For example, it would click on the "Cancel" button even though during the recording we clicked the "Ok" button.

The cause for this incorrect behavior has to do with one of the three approaches to component identification that Pounder uses. If a component has its "name" property set (each subclass of `java.awt.Component` has such a name property, which a programmer can set to tag the component with some string), then Pounder will use that name to find the component inside a given window. Unfortunately, for some standard dialogs, multiple components have the same name. Thus, e.g., when the user clicks on the "Ok" button at recording time, Pounder stores that button's name (say, "button") in its session log. At replay time, it then looks for the first component with the name "button" and happens to find the "Cancel" button.

To work-around this problem, we changed Pounder to test, whenever it tries to use the name to identify a component, whether there are multiple components with that name in the given window. In that case, we use a different component identification approach (the path from the window to the component, represented by the sequence of each child component's index inside its parent component).

This simple fix eliminated the problem and allowed us to correctly replay many Pounder sessions that previously failed.

8.3 Decreasing perturbation

While Figs. 4 and 5 have shown that latency distributions gathered while replaying a session with Pounder are relatively close to the distributions collected when manually performing the sessions, we still wanted to investigate whether we could further quantify the perturbation due to Pounder.

8.3.1 Direct overhead of Pounder player

To determine whether Pounder directly caused a significant computational overhead, we ran Pounder to replay the JHotDraw Draw session on top of a profiler. Our profiler, based on JVMTI Sun Microsystems (2004), periodically samples the call stack of all threads in the Java virtual machine. It thus can measure the approximate amount of time spent in the application and in Pounder.

We used the Trevis (Adamoli and Hauswirth 2010) context tree visualization and analysis framework to combine the collected stack samples into a calling context tree. We found that less than 2% of call stack samples fell into the Pounder player (Pounder's thread that posts events at their appropriate times). Roughly 25% of the 2% overhead was due to translating component-relative coordinates into absolute screen coordinates, which entails a relatively expensive native call. Unfortunately ,that code is necessary, because, no matter

whether mouse events are posted to the OS or the Java event queue, their coordinates have to be translated.

8.3.2 Headless Pounder

Pounder provides a GUI to initiate, observe, and control recording and playback. The presence of that GUI during playback can affect performance, because the GUI itself will, like the application under test, cause GUI events to be dispatched.

We thus developed a command-line recording and replay front-end, which avoids this overhead. However, we did not observe a significant difference in performance between using Pounder's GUI and our command-line front-end.

8.3.3 Asynchronous playback

When posting events to a queue, the events are not usually processed synchronously. Moreover, when using the `Robot` to post events to the OS event queue, the OS or the Java GUI toolkit might reorder, combine, drop, or insert additional events.

We thus instrumented the Pounder player to observe all requests being posted via the `Robot` and to observe all events being dispatched from the Java event queue. Ideally, every event being posted to the OS event queue would be forwarded to the Java event queue and dispatched from there. In our experiment, we found that while most calls to the `Robot` were paired with a subsequent event dispatch, there were some exceptions, which we were unable to explain. We believe that these exceptions are responsible for some of the performance difference we observed.

8.4 Discussion

In this section, we described our investigation into Pounder and the changes we made based on those results. While our new version of Pounder clearly is an improvement over the original tool, we obviously were not able to provide a general solution to component identification, temporal synchronization, and perturbation. Those problems are fundamental, and we believe that there is no approach that can solve all three at once. However, in many situations, for many applications, Pounder should now be robust enough, and its perturbation should be low enough, to allow automated GUI performance tests with actionable results.

9 Case studies

While GUI performance testing is relevant to ensuring the quality of interactive applications, practical and accurate GUI performance testing approaches can have a broader impact, especially in evaluating the platforms and systems on top of which interactive applications are built.

When the computer system community evaluates the performance improvements of their innovations, such as novel compiler optimizations, garbage collection approaches, and improvements to runtime environments, operating systems, or processor microarchitectures, they have to evaluate whether their innovations indeed improve performance. A considerable body of work in the systems community outlines pitfalls and

shortcomings in such performance evaluations, such as issues in the statistical analysis of performance measurements (Georges et al. 2007), bias due to ignoring important factors in experimental setups (Mytkowicz et al. 2009), and bias in commonly used profilers (Mytkowicz et al. 2010). Most importantly, Blackburn et al. (2006) point out that the complex interaction between Java applications and the architecture, compiler, virtual machine, and memory management requires more extensive performance evaluations than what has been done traditionally with C, C++, and Fortran applications. They construct a new suite of Java benchmark applications, called the Dacapo benchmarks, which is now widely used in systems research evaluations. However, in the OOPSLA paper that introduces their Dacapo suite, they write "we excluded GUI applications since they are difficult to benchmark systematically". That statement explicitly expressed what must be a common sentiment in the community, because, to the best of our knowledge, *none* of the widely used benchmark suites includes truly interactive applications.

Given that a large segment of computer systems (desktops, notebooks, tablets, cell phones) are used by end users running interactive programs and given that interactive applications significantly differ from traditional programs in terms of their structural and behavioral characteristics (Zaparanuks and Hauswirth 2010; Brooks et al. (2009)), limiting performance evaluations of systems innovations to non-interactive applications constitutes a bias that may render conclusions drawn from such evaluations invalid. This issue was one of the motivating factors for our work.

In this section, we will use our automatic GUI performance testing approach to study the impact of various system features on perceptible performance. It is not our goal to evaluate specific system features; a thorough study of the performance impact of even a single system feature would require much more extensive experiments. Our goal in this section is to demonstrate how systems researchers and designers can use our approach to augment their benchmark suites with interactive applications and with a way to measure performance as perceived by users. Moreover, in each experiment, we include measurements taken manually, to verify to which degree the test automation infrastructure affects measurement results. Unless otherwise stated, the platforms, applications, interactive sessions, and inputs for the experiments in this section correspond to those we studied in Sect. 6.

9.1 Operating system

On which operating system is a given GUI application more performant? This case study shows how an automatic GUI performance testing approach can help to answer this kind of question. We conducted experiments on two predominant operating systems used by users of interactive applications, Microsoft Windows and Mac OS X. Figure 6 compares the performance of these two systems when running Euclide and GanttProject. The solid lines represent Windows, and the dashed lines represent Mac OS X. The thin lines represent the mean performance during five replays with Pounder, and the thicker lines represent the mean performance of five manually repeated sessions.

The graph for *Euclide* shows a significant difference between operating systems: the perceptible performance is worse on the Mac. It is interesting to note that this difference is the same no matter whether we repeat the interaction manually (thick curves) or whether we record and replay a session with Pounder (thin curves). This means that in this case, Pounder does not affect our conclusion that Euclide runs slower on the Mac than on Windows.

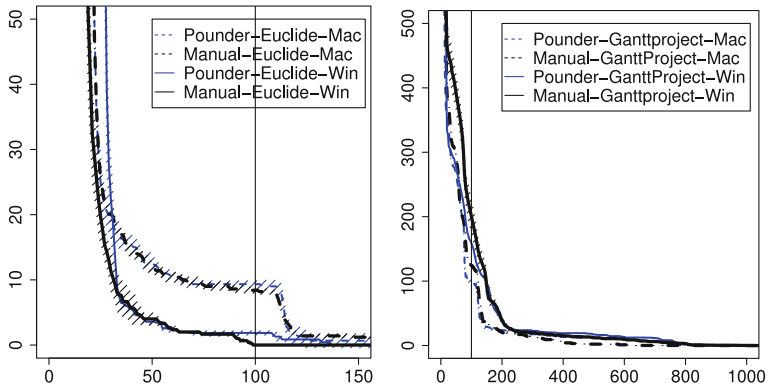

Fig. 6 Automated measurement of impact of operating system on perceptible performance

GanttProject shows a relatively small difference between the Mac and the Windows curves. The interesting region around the knee between 100 and 200 ms shows that GanttProject performs better on the Mac (dashed). Moreover, in that region, the Pounder results (thin) agree quite closely with the manual (thick) results. However, in other regions (e.g., from 250 to 750 ms), the Mac curve for Pounder deviates from the manual curve. Nevertheless, even in this region, Pounder and manual measurements lead to the same conclusion: GanttProject runs slightly faster on the Mac (dashed) than on Windows (solid).

These cases show that the fact that we use Pounder to automatically drive the application, instead of manually interacting with the application, affects the cumulative latency distribution. However, this perturbation of the measurements does not change the conclusions we draw from comparing the distributions: Euclide runs faster on Windows, and GanttProject runs slightly faster on the Mac.

9.2 Client versus server mode

How do virtual machine features affect the performance of interactive applications? This case study provides an example of how a virtual machine researcher could use automated GUI performance testing to evaluate the benefits of her innovations.

The HotSpot virtual machine that is part of Oracle's and Apple's Java distribution includes two main configurations: client mode and server mode. The client mode is optimized toward running interactive applications. Here, we use our approach to evaluate whether we can see a difference in interactive performance between the two modes.

Figure 7 shows the performance of JHotDraw Draw and Jmol on the client and the server virtual machine configuration. For JHotDraw Draw, we see little difference between client (dashed) and server (solid) configuration. For Jmol, though, the experiment shows a significant difference: the server configuration reduces the latency of a significant number of events from roughly 350 to 250 ms. For both applications, the Pounder curves differ from the manual curves. However, in both cases, the conclusions drawn from the Pounder curves are the same as those drawn from the manual curves.

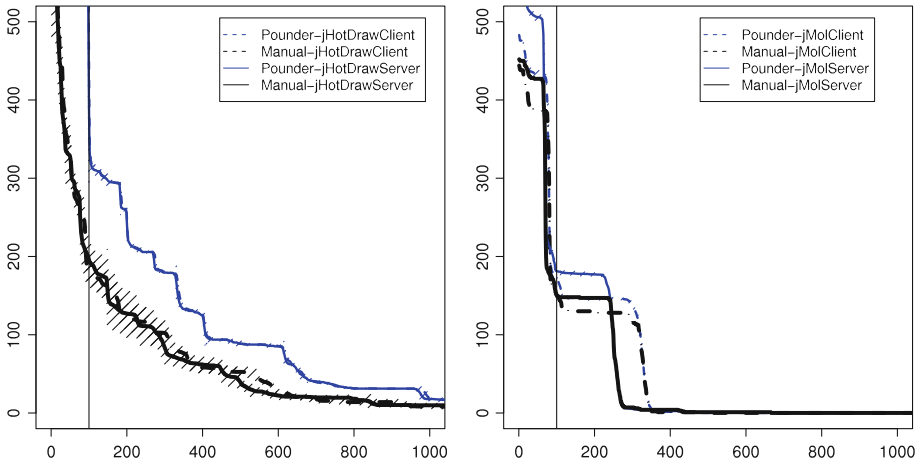

Fig. 7 Automated measurement of impact of client versus server virtual machine on perceptible performance

9.3 Heap size

How does the memory management approach affect interactive performance? This case study represents an example use of automatic GUI performance testing to show how the size of the heap, a key parameter of any memory management approach, can affect interactive performance.

Figure 8 shows the performance of JHotDraw Draw with different heap sizes (JHotDraw Draw with 40, 64, and 128 MB; Jmol with 50, 64, and 1,024 MB). In our scenarios, the heap size does not significantly affect the perceptible performance. Again, Pounder and manual replay lead to the same conclusion.

9.4 Input size

The previous three case studies represented examples of performance experiments of interest to systems builders. This case study highlights a relevant parameter in any GUI performance testing experiment: the size of the artifact (e.g., document, table, diagram, and video) the user is manipulating in the application. Many interactive applications essentially are viewers or editors for some kind of artifact. The size of the artifact can affect the application's perceptible performance in two ways: first, it affects the amount of memory used by the application and thus the cost of memory management, and second, it affects the amount of time needed by the application's algorithms that operate on the artifact.

Figure 9 shows how the performance of JHotDraw Draw and Jmol depends on the size of their inputs. We used inputs of two or three significantly different sizes. This approach is common to performance benchmarking in batch applications. For JHotDraw, the inputs represent realistically complex diagrams we created ourselves and for Jmol we used two realistic molecules that are part of the Jmol distribution.

The three different inputs for *JHotDraw Draw* represent drawings with 400 shapes (large), 200 shapes (medium), and 50 shapes (small). We automatically generated these three drawings so that their structure allowed a single interaction script to meaningfully operate on any of the three drawings. Figure 9 clearly shows that JHotDraw's performance

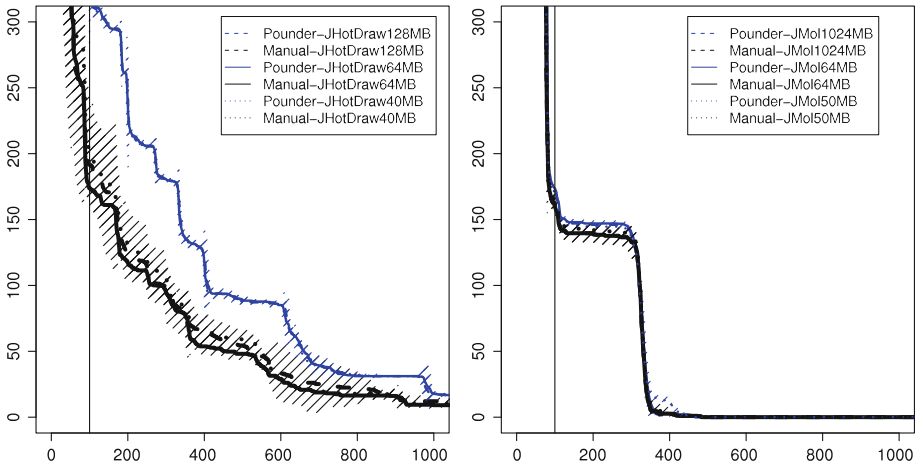

Fig. 8 Automated measurement of impact of heap size on perceptible performance

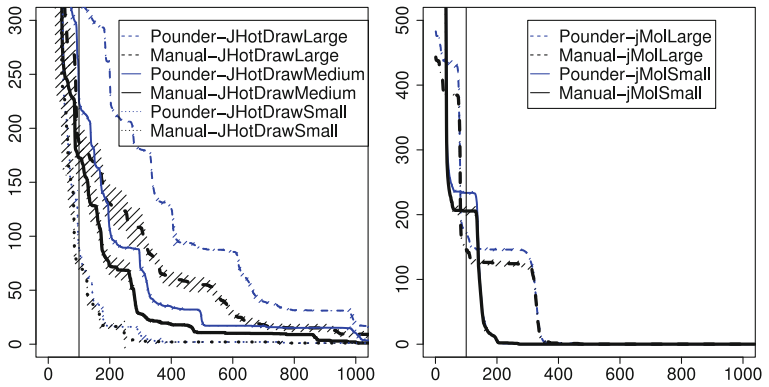

Fig. 9 Automated measurement of impact of input size on perceptible performance

on the large inputs (dashed lines) is worse than on the medium inputs (solid lines), which is worse than the performance on the small inputs (dotted lines). This relationship is true no matter whether we manually repeat the interaction (thick lines) or use Pounder for capture and replay (thin lines).

Jmol visualizes and animates chemical molecules it loads from a file. We used an alpha helix (20 kB file) as a small input and a hemoglobin molecule (548 kB) as a large input. Figure 9 shows that visualizing the large molecule (dashed line) generally is slower. However, between 90 and 140 ms, the small input curves stay *above* the large input curves. The cause for this seemingly anomalous artifact is that the big molecule takes longer to paint, which leads to a lower frame-rate during its animation. Thus, Jmol will perform *fewer* repaints over the course of the interaction, which will shift the big molecule's curve down. Each of the repaints, however, will take longer. This will shift the big molecule's curve to the right. The downward shift of the big molecule's curve is what causes the crossover between 90 and 140 ms. The fact that this artifact shows up in both the manual

and the Pounder curves shows the faithfulness of Pounder-based performance characterizations.

9.5 Program version

Our first three case studies used capture and replay tools for system performance evaluation. In this case study, we focus on the more traditional use for such tools: the testing of the performance of the actual interactive application. In particular, we show that our approach can detect differences in performance between two versions of the same application. Such differences can occur when developers *introduce* or *fix* a performance regression. In a regression testing scenario, the performance of each new version of an application is evaluated and compared with the prior version's performance. Significant performance degradations are considered performance bugs and will have to be fixed before releasing the new version. Performance regression testing, like functional regression testing, benefits from automation. To automate GUI performance regression testing, though, we require the automation of user interactions. This is exactly what a GUI capture and replay approach provides: we can record an interactive session once and automatically replay it on subsequent versions of the application. We now demonstrate this use case on two subsequent versions of JFreeChart. Note that it is not the goal of this case study to show how to find the cause of a performance bug, we just want to show that capture and replay approaches can be used to detect differences (in this scenario, an improvement due to a bug fix) in perceptible performance *across program versions*.

Figure 10 shows the performance of the old (1.0.12, solid line) and new (1.0.13, dashed line) version of JFreeChart. The curves show that the new version is faster than the old version. This finding applies to the manually performed measurements (thick lines) as well as the measurements using Pounder (thin) for record and replay.

The fact that there is a significant performance difference between these two versions of JFreeChart is not a coincidence: we picked these versions because version 1.0.13 includes a fix for a performance bug existing in prior versions. That bug significantly impacts the responsiveness when rendering large time charts. It is one of the goals of performance regression test automation to help detect such performance bugs.

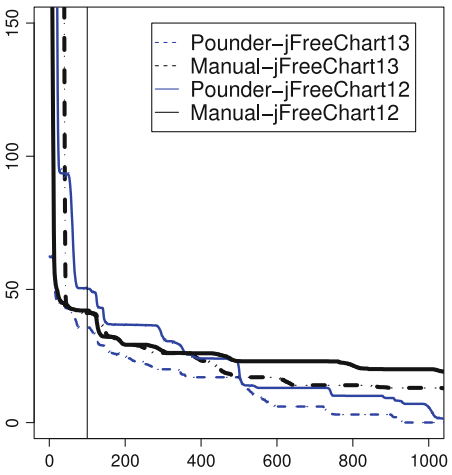

Fig. 10 Automated measurement of perceptible performance across program versions

9.6 Discussion

In all the above case studies, we have seen that the conclusions drawn from measurements based on automated replay with Pounder corresponded to the conclusions based on repeated manual interactive sessions. While the use of Pounder clearly perturbed the latency distributions, it did not affect any of the conclusions.

Moreover, as all the case study figures show, the confidence intervals of the manually repeated runs (thick curves) are often much wider than the intervals of the runs replayed by Pounder (thin curves). This is due to Pounder being more deterministic in replaying a session than a human user who tries to repeat his interactions. Thus, besides saving time (due to the automation of replay), this approach also improves the comparability of the different runs.

10 Threats to validity

Our findings are based on the study of five open-source capture and 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 any possible solution would significantly perturb performance.

A second 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, which are necessarily time limited. Limiting session durations could prevent us from discovering performance problems that only manifest themselves in long sessions, such as gradual slowdowns due to memory leaks. 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.

Finally, our experiments focus on a single hardware platform, on only two operating systems and on only one kind of virtual machine (HotSpot). Our performance measurements are thus specific to these environments, and our finding that the use of Pounder does not affect the conclusions drawn from performance measurements might not hold on other platforms. However, the methodology we present in this paper provides a guideline that researchers and developers can follow to evaluate system or application performance on their own platforms. Moreover, our the improvements to the most practical GUI testing tool, Pounder, increase its robustness no matter which platform it is running on.

11 Related work

In this section, we discuss related work that goes beyond the survey in Sect. 3. We focus on work specifically discussing the problem of replaying user interactions and on work about the general concept of deterministically replaying program executions.

11.1 Replaying user interactions

Reliably capturing and replaying user interactions is difficult. McMaster and Memon (2009) 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 (Steven et al. 2000) 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. (2009) 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 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 (2008) 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.

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.

11.2 Low-level record/replay approaches

Ronsse et al. (2003) provide a general description of using record/replay approaches for non-deterministic program executions. Their view is informed by their prior work on RecPlay (Ronsse and De Bosschere 1999), 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 and 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 (Liu et al. 2009), postmortem debugging (Narayanasamy et al. 2005), and intrusion detection (de Oliveira et al. 2006). All these approaches operate on a much lower level of abstraction, recording architectural events about program execution. The higher abstraction level of GUI capture and 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.

12 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 and 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 and 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. 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. We have improved Pounder's reliability and decreased its perturbation based on our investigations.

Finally, our five case studies show that Pounder, even though it slightly perturbs performance measurements, does not affect the conclusions based on those measurements and thus is a valuable tool for reducing the cost of performance testing.

Acknowledgments This work has been conducted in the context of the Binary Translation and Virtualization cluster of the EU HiPEAC Network of Excellence. It has been funded by the Swiss National Science Foundation under grant number 125259.

References

Adamoli, A., & Hauswirth, M. (2010). Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *SoftVis '10: Proceedings of the ACM symposium on software visualization*.

Alsmadi, I. (2008). 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)* (pp. 581–585). Washington, DC, USA: IEEE Computer Society.

Belli, F. (2001). Finite-state testing and analysis of graphical user interfaces. *Software reliability engineering, international symposium on*, 0:34.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., et al. (2006). The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications* (pp. 169–190). New York, NY, USA: ACM.

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

Brooks, P. A., Robinson, B. P., & Memon, A. M. (2009). An initial characterization of industrial graphical user interface systems. *Software testing, verification, and validation, 2008 international conference on*, 0:11–20.

Chang, T.-H., Yeh, T., & Miller, R. C. (2010). Gui testing using computer vision. In *Proceedings of the 28th international conference on human factors in computing systems, CHI '10* (pp. 1535–1544). New York, NY, USA: ACM.

Chinnapongse, V., Lee, I., Sokolsky, O., Wang, S., & Jones, P. L. (2009). Model-based testing of gui-driven applications. In *Proceedings of the 7th IFIP WG 10.2 international workshop on software technologies for embedded and ubiquitous systems, SEUS '09* (pp. 203–214). Berlin, Heidelberg: Springer-Verlag.

de Oliveira, D. A. S., Crandall, J. R., Wassermann, G., Felix Wu, S., Su, Z., & Chong, F. T. (2006). 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* (pp. 66–71). New York, NY, USA: ACM.

Deursen, A.,, & Mesbah, A. (2010). Research issues in the automated testing of ajax applications. In *Proceedings of the 36th conference on current trends in theory and practice of computer science, SOFSEM '10* (pp. 16–28). Berlin, Heidelberg: Springer-Verlag.

El Ariss, O., Xu, D., Dandey, S., Vender, B., McClean, P., & Slator, B. (2010). A systematic capture and replay strategy for testing complex gui based java applications. In *Proceedings of the 2010 seventh international conference on information technology: New generations, ITNG '10* (pp. 1038–1043). Washington, DC, USA: IEEE Computer Society.

Elbaum, S., Karre, S., & Rothermel, G. (2003). Improving web application testing with user session data. In *Proceedings of the 25th international conference on software engineering, ICSE '03* (pp. 49–59). Washington, DC, USA: IEEE Computer Society.

Elbaum, S., Rothermel, G., Karre, S., & Fisher, M., II. (2005). Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering, 31*, 187–202.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems and applications* (pp. 57–76). New York, NY, USA: ACM.

Grechanik, M., Xie, Q., Fu, C. (2009). Maintaining and evolving gui-directed test scripts. In *ICSE '09: Proceedings of the 2009 IEEE 31st international conference on software engineering* (pp. 408–418). Washington, DC, USA: IEEE Computer Society.

Hackner, D. R., & Memon, A. M. (2008). Test case generator for guitar. In *Companion of the 30th international conference on software engineering, ICSE Companion '08* (pp. 959–960). New York, NY, USA: ACM.

Jovic, M., Adamoli, A., Zaparanuks, D., & Hauswirth, M. (2010) Automating performance testing of interactive java applications. In: *AST '10: Proceedings of the 5th Workshop on Automation of Software Test*, pp. 8–15, New York, NY, USA, 2010. ACM.

Jovic, M., & Hauswirth, M. (2008). 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* (pp. 137–146). New York, NY, USA: ACM.

Kasik, D. J., & George, H. G. (1996). Toward automatic generation of novice user test scripts. In *Proceedings of the SIGCHI conference on human factors in computing systems: Common ground, CHI '96* (pp. 244–251). New York, NY, USA: ACM.

Li, P., Huynh, T., Reformat, M., & Miller, J. (2007). A practical approach to testing gui systems. *Empirical Software Engineering, 12*, 331–357.

Li, K., Wu, M. (2004). *Effective GUI testing automation: Developing an automated GUI testing tool*. Alameda, CA, USA: SYBEX Inc.

Lindvall, M., Rus, I., Donzelli, P., Memon, A., Zelkowitz, M., Betin-Can, A, et al. (2007). Experimenting with software testbeds for evaluating new technologies. *Empirical Software Engineering: An International Journal, 12*(4), 417–444.

Liu, H., Jin, H., Liao, X., Hu, L., & Yu, C. (2009). 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* (pp. 101–110). New York, NY, USA: ACM.

Liu, C.-H., Kung, D. C., Hsia, P., Hsu, C.-T. (2000a). Object-based data flow testing of web applications. In *Proceedings of the the first Asia-Pacific conference on quality software (APAQS'00), APAQS '00* (pp. 7–16). Washington, DC, USA: IEEE Computer Society.

Liu, C.-H., Kung, D. C., Hsia, P., & Hsu, C.-T. (2000b). Structural testing of web applications. In *Proceedings of the 11th international symposium on software reliability engineering* (pp. 84–96). Washington, DC, USA: IEEE Computer Society.

Lowell, C., & Stell-Smith, J. (2003) Successful automation of gui driven acceptance testing. In *Proceedings of the 4th international conference on extreme programming and agile processes in software engineering, XP'03* (pp. 331–333). Berlin, Heidelberg: Springer-Verlag.

Lucca, G. D., Fasolino, A., & Faralli, F. (2002). Testing web applications. In *Proceedings of the international conference on software maintenance (ICSM'02)* (pp. 310–319). Washington, DC, USA: IEEE Computer Society.

Marchetto, A., Ricca, F., & Tonella, P. (2008a). A case study-based comparison of web testing techniques applied to ajax web applications. *International Journal of Software Tools and Technology Transactions, 10*, 477–492.

Marchetto, A., Tonella, P., & Ricca, F. (2008b). State-based testing of ajax web applications. In *ICST* (pp. 121–130). IEEE Computer Society.

McMaster, S., & Memon, A. (2008). Call-stack coverage for gui test suite reduction. *ACM Transactions of Software Engineering, 34*, 99–115.

McMaster, S., & Memon, A. M. (2009). 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*.

Memon, A. M. (2008). Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions of Software Engineering Methodology, 18*(2), 1–36.

Memon, A. M., Pollack, M. E., & Soffa, M. L. (2001) Hierarchical gui test case generation using automated planning. *IEEE Transactions of Software Engineering, 27*, 144–155.

Memon, A., Nagarajan, A., & Xie, Q. (2005). Automating regression testing for evolving gui software. *Journal of Software Maintenance, 17*, 27–64.

Memon, A. M., & Xie, Q. (2005). Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Transactions of Software Engineering, 31*, 884–896.

Mesbah, A. , & van Deursen, A. (2009). Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st international conference on software engineering, ICSE '09* (pp. 210–220). Washington, DC, USA: IEEE Computer Society.

Meszaros, G. (2003). Agile regression testing using record & playback. In *Companion of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA '03* (pp. 353–360). New York, NY, USA: ACM.

Mitchell, A., & James F. Power. An approach to quantifying the run-time behaviour of java gui applications. In: *WISICT '04: Proceedings of the winter international symposium on Information and communication technologies*, pp. 1–6. Trinity College Dublin, 2004.

Mu, B, Zhan, M., & Hu, L. (2009). Design and implementation of gui automated testing framework based on xml. In *Proceedings of the 2009 WRI world congress on software engineering—Vol. 04, WCSE '09* (pp. 194–199). Washington, DC, USA: IEEE Computer Society.

Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In: *ASPLOS '09: Proceeding of the 14th international conference on architectural support for programming languages and operating systems* (pp. 265–276). New York, NY, USA: ACM.

Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2010). Evaluating the accuracy of java profilers. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on programming language design and implementation* (pp. 187–197). New York, NY, USA: ACM.

Narayanasamy, S., Pokam, G., & Calder, B. (2005). Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual international symposium on computer architecture* (pp. 284–295). Washington, DC, USA: IEEE Computer Society.

Nguyen, D. H., Strooper, P., & Suess, J. G. (2010). Model-based testing of multiple gui variants using the gui test generator. In *Proceedings of the 5th workshop on automation of software test, AST '10* (pp. 24–30). New York, NY, USA: ACM.

Ricca, F., & Tonella, P. (2001). Analysis and testing of web applications. In *Proceedings of the 23rd international conference on software engineering, ICSE '01* (pp. 25–34). Washington, DC, USA: IEEE Computer Society.

Ronsse, M., & Bosschere, K. D. (1999). Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer System, 17*(2), 133–152.

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

Ruiz, A., & Price, Y. W. (2007). Test-driven gui development with testng and abbot. *IEEE Software*, 24, 51–57.

Ruiz, A., & Price, Y. W. (2008). Gui testing made easy. In *Proceedings of the testing: Academic & industrial conference—practice and research techniques* (pp. 99–103). Washington, DC, USA: IEEE Computer Society.

Sampath, S. (2004). Towards defining and exploiting similarities in web application use cases through user session analysis. In *Proceedings of the second international workshop on dynamic analysis*.

Shehady, R. K., & Siewiorek, D. P. (1997). A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th international symposium on fault-tolerant computing (FTCS '97), FTCS '97* (p. 80). Washington, DC, USA: IEEE Computer Society.

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

Strecker, J., & Memon, A. M. (2008). Relationships between test suites, faults, and fault detection in gui testing. In *ICST '08: Proceedings of the first international conference on software testing, verification, and validation*. Washington, DC, USA: IEEE Computer Society.

Sun, Y., & Jones, E. L. (2004). Specification-driven automated testing of gui-based java programs. In *Proceedings of the 42nd annual Southeast regional conference, ACM-SE 42* (pp. 140–145). New York, NY, USA: ACM.

Sun Microsystems. (2004). Java Virtual Machine Tool Interface (JVMTI), http://www.java.sun.com/j2se/1.5.0/docs/guide/jvmti.

Silva, J. C., Saraiva, J., & Campos, J. C. (2009). A generic library for gui reasoning and testing. In *Proceedings of the 2009 ACM symposium on applied computing, SAC '09* (pp. 121–128). New York, NY, USA: ACM.

White, L., & Almezen, H. (2000). Generating test cases for gui responsibilities using complete interaction sequences. *Software reliability engineering, international symposium on*, 0:110.

Xie, Q. (2006). Developing cost-effective model-based techniques for gui testing. In *Proceedings of the 28th international conference on software engineering, ICSE '06* (pp. 997–1000), New York, NY, USA: ACM.

Xie, Q., & Memon, A. M. (2007). Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions of Software Engineering Methodology*.

Xie, Q., & Memon, A. M. (2008) Using a pilot study to derive a gui model for automated testing. *ACM Transactions of Software Engineering Methodology, 18*, 7:1–7:35.

Yuan, X., Cohen, M. B., & Memon, A. M. (2009). Towards dynamic adaptive automated test generation for graphical user interfaces. In *Proceedings of the IEEE international conference on software testing, verification, and validation workshops* (pp. 263–266). Washington, DC, USA, 2009. IEEE Computer Society.

Yang, J.-T., Huang, J.-L., Wang, F.-J., & Chu, W. C. (1999). An object-oriented architecture supporting web application testing. In *23rd international computer software and applications conference, COMPSAC '99* (pp. 122–127). Washington, DC, USA: IEEE Computer Society.

Yuan, X., & Memon, A. M. (2007). Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th international conference on software engineering* (pp. 396–405). Washington, DC, USA: IEEE Computer Society.

Yuan, X., & Memon, A. M. (2010). Generating event sequence-based test cases using gui runtime state feedback. *IEEE Transactions on Software Engineering, 36*, 81–95.

Zaparanuks, D., & Hauswirth, M. (2010). Characterizing the design and performance of interactive java applications. In *ISPASS* (pp. 23–32). IEEE Computer Society.

Author Biographies

Andrea Adamoli is a Ph.D. student at the Faculty of Informatics at the University of Lugano. His research mostly focuses on performance benchmarking and analysis. Adamoli received his Master of Science in Embbeded Systems Design from the ALaRI Institute located at the University of Lugano and affiliated with ETH Zürich and Politecnico of Milan.

Dmitrijs Zaparanuks is a Ph.D. student at the Faculty of Informatics at the University of Lugano. His research combines performance measurements and design analysis of a program. Zaparanuks received his Master of Science in Embbeded Systems Design from the ALaRI Institute located at the University of Lugano and affiliated with ETH Zürich and Politecnico of Milan.

Milan Jovic is a Ph.D. student at Faculty of Informatics at the University of Lugano. His research is mostly focused on interactive application performance analysis. Jovic received his Masters degree from the Faculty of Electrical Engineering at the University of Nis in Serbia.

Matthias Hauswirth is an assistant professor at the Faculty of Informatics at the University of Lugano. He is interested in performance measurement, understanding, and optimization. Hauswirth received his Ph.D. from the University of Colorado at Boulder.