JETracer - A Framework for Java GUI Event Tracing

Arthur-Jozsef Molnar, PhD.

Babes-Bolyai University
Cluj-Napoca, Romania

arthur@cs.ubbcluj.ro
Overview

1 What is JETracer?

2 Motivation

3 Related work
   - Valgrind
   - Oracle Java tools
   - JIVE and JOVE tools
   - Squish for Java

4 JETracer
   - Architecture
   - Sample Observer Application
   - Empirical Evaluation

5 Future Work
What is JETracer?

**Java Event Tracer** - a framework that allows capturing GUI information for applications developed using the AWT, Swing or SWT toolkits

- Allows recording as well as capturing event information in real time
- Transparent to the target application and the GUI toolkit used
- Easy to deploy - interested applications must implement a one method listener interface
- Easy to extend for other GUI toolkits
- Open sourced under EPL (or other, on request), tutorial and examples available at [https://bitbucket.org/arthur486/jetracer](https://bitbucket.org/arthur486/jetracer)
- Works under both Windows and Linux, x86 & x86-64
GUI code consists of up to 50% of all application code [Memon, 2001]

How can software tools help practitioners deal with GUI application lifecycle?
  - Analysis and design
  - Prototyping
  - GUI testing
  - Program visualization, comprehension

Which of the existing tools are open-source?
Industry survey over 1400 professional developers [Maalej, 2014]

- The strategies, tools and problems when comprehending software
- Significant findings
  - Developers interact with the GUI in order to find the starting point of further interaction
  - IDE features that help program comprehension are not known or not used by developers
  - ”Developers prefer informal, pragmatic, integrated solutions”
  - ”Industry developers do not use dedicated program comprehension tools developed by the research community”
Tools that have inspired or are related to JETracer

- Valgrind
- Oracle Java Flight Recorder and Java Mission Control
- JIVE and JOVE
- Squish for Java
The Valgrind suite

- Open source tool suite for detecting C/C++ memory & threading bugs
- Runs on several platforms, easy to deploy
- Uses dynamic binary instrumentation, transparent to target application
- Widely used & suitable for any type of software

In implementation and operation, Valgrind and JETracer share a common philosophy

---

http://valgrind.org/
Related work - Oracle Java tools

**Oracle Java Mission Control**
- Monitor the Java VM parameters (CPU, heap, etc.)

**Oracle Java Flight Recorder**
- Collect events at OS, Java or application levels (even in production environment)
- Allows thread sync, I/O, GC and method profiling
- Useful visualizations
Related work - Oracle Java tools

Figure: Oracle Java Mission Control

http://hirt.se/blog/?p=343
Related work - JIVE and JOVE

**JIVE and JOVE - Java as it happens** [Reiss, 2005]
- Dynamic software visualizer that works at statement level
- Low overhead using basic block counting
- Maps statistical properties to graphical visualization
- Uses code instrumentation on the target application
The Squish for Java toolset

- Includes support for AWT, SWT, Swing, JavaFX controls
- Uses code instrumentation on the target application
- Easy test case recording & replay across multiple platforms
- We used it when (performance) testing JETracer - the scripts we used are available on JETracer website
- Proprietary implementation

http://www.froglogic.com/
Java Event Tracer - a framework that allows capturing GUI information for applications developed using the AWT, Swing or SWT toolkits

- Available under friendly open-source license
- Transparent to the target application
- Easy to use - you only need to implement a listener having one method
- Easy to deploy - agent module on target application classpath, host module on observer application classpath
- Can be extended by implementing a new agent module - boilerplate code already written!
- Tested under Windows 7,8 and Ubuntu Linux (both x86 & x86-64) using Oracle Java and OpenJDK
Figure: Modules in a JETracer deployment
Java Event Tracer - architecture

- **Host & Agent Module** - JETracer framework
- **GUI Toolkit** - currently one of AWT, Swing or SWT
- **Target Application** - we want the GUI events from this application
- **Observer Application** - analysis, visualization or testing application
  - instance of `EventMessageListener` implemented within Observer Application which receives event objects
Host Module - roles

1. Manage communication with agent
2. Forward event information to subscribed listeners

Using the Host Module

- Must be on Observer Application classpath
- Connecting to an agent within observer application

```java
InstrService is = new InstrService();
InstrConnector connector = is.configure(config);
connector.connect();
connector.addEventMessageListener(this);
```

- The EventMessageListener

```java
public interface EventMessageListener extends EventListener {
    public void messageReceived(EventMessageReceivedEvent event);
}
```
Host Module - configuration

- **Event granularity** - *all* events, or just events *handled* by the target application
- **Event filter** - used to ignore certain events entirely (e.g. mouse move).
- **Target Application** - we want the GUI events from this application
- **Observer Application** - analysis, visualization or testing application
  - *l* - instance of *EventMessageListener* implemented within Observer Application which receives event objects
Host Module - receiving events

- *EventMessage* wrapped in an *EventMessageReceivedEvent*
- Event information captured
  - **Id** - unique for each event
  - **Widget information** - class, size as well as location on screen and within parent container
  - **Screenshot** - of application window where the control that fired the event is located
  - **Event information** - event type, associated listeners, timers
Agent Module - roles

1. Instrument required classes
2. Record events synchronously when fired
3. Handle communication with host

New implementations - must handle 1 & 2

Using the Agent Module

- Must be on target application boot classpath
- Target application started with agent module as Java Agent
Agent Module - current flavours

- AWT/Swing
  - Support for both high&low level events
  - Implementation was laborious due to each component having a listener queue

- SWT
  - Required instrumenting the EventTable class

Detailed list of events that can be traced available on our website

4 https://bitbucket.org/arthur486/jetracer
Sample Observer Application - getting started with JETracer quickly

- A sample Observer Application to illustrate how to use JETracer
- Available on our website
- Simple GUI that shows recorded event timeline together with captured screenshots
Figure: Sample Observer Application tracing FreeMind events
Empirical Evaluation

Evaluation areas

1. Deploying JETracer for complex, popular open-source software
2. Objective measurement of performance using automated interaction scenarios
3. Perceived impact on application performance
Empirical Evaluation - Deployment

**Deployment** - We searched for open source software fulfilling several criteria

1. Cover most, if not all GUI controls is AWT, Swing and SWT
2. Have an active user base and be in active development
3. Easy to set up and tear down, in order to ensure test repeatability

**Applications used** in our evaluation

- **AWT/Swing**: aTunes 3.1.2, FreeMind 1.0.1, jEdit 4.5.2
- **SWT**: Azureus 2.0.4.0 and TuxGuitar 1.0.2
Empirical Evaluation - Measuring Performance

Evaluation process

- We created automated scenarios that mimic real-life user interaction for each application using Squish for Java
- Made sure results were consistent cross-platform
- Data was transmitted locally over socket to factor out network latency
- Hardware used: a modern PC with Windows 8.1

Factors found to affect performance

- Event granularity - recording all or just handled events
- Recording screenshots - yes/no

We replayed each scenario 4 times to test all combinations
**Interaction scenarios**, synthesized by number of fired events

<table>
<thead>
<tr>
<th>Application</th>
<th>All Events</th>
<th>Handled Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes 3.1.2</td>
<td>155,502</td>
<td>1,724</td>
</tr>
<tr>
<td>Azureus 2.0.4.0</td>
<td>11,149</td>
<td>230</td>
</tr>
<tr>
<td>FreeMind 1.0.1</td>
<td>356,762</td>
<td>5,308</td>
</tr>
<tr>
<td>jEdit 4.5.2</td>
<td>38,708</td>
<td>1,940</td>
</tr>
<tr>
<td>TuxGuitar 1.0.2</td>
<td>13,696</td>
<td>1,802</td>
</tr>
<tr>
<td>TOTAL</td>
<td>575,817</td>
<td>11,004</td>
</tr>
</tbody>
</table>
Table: Average overhead per event without screenshot recording (in milliseconds).

<table>
<thead>
<tr>
<th>Event granularity</th>
<th>All</th>
<th>Handled</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes 3.1.2</td>
<td>0.09</td>
<td>0.21</td>
</tr>
<tr>
<td>Azureus 2.0.4.0</td>
<td>0.13</td>
<td>0.31</td>
</tr>
<tr>
<td>FreeMind 1.0.1</td>
<td>0.09</td>
<td>0.18</td>
</tr>
<tr>
<td>jEdit 4.5.2</td>
<td>0.11</td>
<td>0.18</td>
</tr>
<tr>
<td>TuxGuitar 1.0.2</td>
<td>0.13</td>
<td>0.22</td>
</tr>
</tbody>
</table>

- We obtained consistently low overhead
- Overhead for *handled* events is higher due to reflection calls to retrieve listener classes
- With **no screenshot** recording, application performance is subjectively non distinguishable from vanilla run
Table: Average overhead per event with screenshot recording (in milliseconds).

<table>
<thead>
<tr>
<th>Event granularity</th>
<th>All</th>
<th>Handled</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes 3.1.2</td>
<td>1.78</td>
<td>23.21</td>
</tr>
<tr>
<td>Azureus 2.0.4.0</td>
<td>31.77</td>
<td>34.07</td>
</tr>
<tr>
<td>FreeMind 1.0.1</td>
<td>2.07</td>
<td>27.95</td>
</tr>
<tr>
<td>jEdit 4.5.2</td>
<td>10.10</td>
<td>31.09</td>
</tr>
<tr>
<td>TuxGuitar 1.0.2</td>
<td>28.02</td>
<td>31.96</td>
</tr>
</tbody>
</table>

- Overhead consistently higher and not so consistent (events before GUI is visible, different windows sizes to record)
- When **recording screenshots** for **handled** events, overhead is consistent
Figure: Distribution of incurred overhead (milliseconds) when tracing handled FreeMind events

- Overhead clumped in $< 14ms$ (GUI not visible) or $48 – 80ms$ range
- Can estimate overhead if event firing frequency is known ahead of time
Measured performance

- Broadly usable without screenshot recording
- Usable to track handled events while recording screenshots

Perceived performance impact

- No observable impact without screenshot recording
- Moderate observable impact when recording screenshots for handled events
- Severe impact when recording screenshots for all events
Future Work

- Lower the impact of recording screenshots
- Investigate recording coverage information
- Extend the framework to new toolkits such as JavaFX
- Investigate the use on mobile platforms
- Ensure its visibility in the open-source tool developer community
Future Work

https://bitbucket.org/arthur486/jetracer
Motivation

Graphical User Interface (GUI) applications are ubiquitous today

- **1992** - Microsoft Windows 3.10/11/12
- **1997** - Nokia Symbian released (as EPOC32)
- **1998** - Google Inc. founded
- **2007** - iPhone released
- **2012** - Windows 8 released
- **2015+** - smartphones, tablets, wearables, mHealth devices? - a trend that is set to continue for the foreseeable future
Related work - GUITAR framework

**The GUITAR framework**

- Open source, developed at University of Maryland
- Main components: GUIRipper, Test Case Generator, Test Case Replayer [Nguyen, 2013]
- Great inspiration for JETracer and target of future integration efforts

---

**Figure:** GUITAR process

![GUITAR process diagram](http://sourceforge.net/projects/guitar/)

---

5http://sourceforge.net/projects/guitar/
Maalej, W. et al.
On the comprehension of program comprehension

Atif Muhammed Memon
A comprehensive framework for testing graphical user interfaces
*PhD Thesis 2001*

Reiss, Steven P. and Renieris, Manos
JOVE: Java As It Happens
*Proceedings of the 2005 ACM Symposium on Software Visualization, 115–124, 2005*

Bao N. Nguyen, Bryan Robbins, Ishan Banerjee and Atif Memon
GUITAR: an innovative tool for automated testing of GUI-driven software
*Automated Software Engineering, 1–41, 2013*