Live Visualization of GUI Application Code Coverage with GUITracer

Arthur-Jozsef Molnar
Faculty of Mathematics and Computer Science
University of Babes-Bolyai
Cluj-Napoca, Romania
arthur@cs.ubbcluj.ro

Abstract—The present paper introduces the initial implementation of a software exploration tool targeting graphical user interface (GUI) driven applications. GUITracer facilitates the comprehension of GUI-driven applications by starting from their most conspicuous artefact - the user interface itself. The current implementation of the tool can be used with any Java-based target application that employs one of the AWT, Swing or SWT toolkits. The tool transparently instruments the target application and provides real time information about the GUI events fired. For each event, call relations within the application are displayed at method, class or package level, together with detailed coverage information. The tool facilitates feature location, program comprehension as well as GUI test creation by revealing the link between the application's GUI and its underlying code. As such, GUITracer is intended for software practitioners developing or maintaining GUI-driven applications. We believe our tool to be especially useful for entry-level practitioners as well as students seeking to understand complex GUI-driven software systems. The present paper details the rationale as well as the technical implementation of the tool. As a proof-of-concept implementation, we also discuss further development that can lead to our tool's integration into a software development workflow.

I. INTRODUCTION

Software tools can help practitioners in virtually all activities undertaken during the life of software starting from requirements analysis to development, program comprehension as well as test case design and execution. When studying how complex IDE's such as Eclipse evolve [1], [2], we observe that newer versions ship with increasingly complex tools for aiding professionals build higher quality software faster. Modern IDE's feature tools for working with artefacts such as UML, code generation and navigation as well as supporting many common development tasks. However, we find that in most cases these tools are centred on the executable representation of the program, namely its source code and associated artefacts, limiting provided functionalities to those directly related to source code.

Our goal is to leverage the latest results from research and industry in order to build new and useful tools for practitioners working on GUI-driven software. Our choice of this field is motivated by the fact that the GUI is the most pervasive paradigm for human-computer interaction, employed by many desktop and mobile applications. In addition, according to [3], in many cases GUI related code takes up to 50% of application code, making it an even more compelling target.

The role of tooling is already established in the literature. The authors of [4] conducted a survey covering over 1400 software professionals who were inquired about their strategies, tools and the problems they encountered when comprehending software. The most significant findings show that most developers employ a white-box strategy for program comprehension. They interact with the application GUI to locate corresponding event handlers in code and in many cases use the IDE in combination with more specialized tools. Of particular note is the finding that "industry developers do not use dedicated program comprehension tools developed by the research community" [4]. The purpose of our work is to provide innovative open-source tools that practitioners have a real need for and that can be used both within the academia as well as industry. After our previously developed JETracer framework [5], [6], the GUITracer tool serves as the next natural step of this strategy.

This paper is structured as follows: the next section introduces the theoretical framework GUITracer is based on, while Section 3 discusses the implementation and features of the tool. The following sections discuss related work and present our conclusions as well as future work.

II. PREREQUISITES

We start from the GUI's characterization as a "hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events" [3]. As provided by Memon in [3], interaction with a GUI application can be modelled as a sequence of events. Industry studies such as [4] show that practitioners approach program comprehension and feature localization tasks at the GUI level. Thus, we consider it beneficial to develop software tools that support this approach. Our goal for GUITracer is to provide a navigable relation between the target application's GUI and its underlying code. We achieve this by providing information regarding how the GUI and the underlying code are related as well as the source code that actually runs once a GUI event is fired. The following sections introduce some theoretical notions that are used within our tool as well as our previously developed JETracer framework, on which our tool is based.

A. Code Coverage for GUI Events

While the literature abounds with code coverage related techniques and tools, we find far less work focused on metrics tailored for GUI-driven applications. In this section we propose several metrics that measure the relation between GUI events and the application source code that handles them. Our approach combines established coverage criteria with Memon's event-sequence coverage defined in [7]. To this we add knowledge gained from call graphs built using static analysis, which improve the code coverage picture. The first step in our effort is to define a GUI event's call graph:

Definition. Event call graph. *Given a GUI event e, we define its event call graph as a subgraph of the application's statically computed call graph that consists of event e's application-level event handlers and all application methods reachable from them.*

The event call graph provides information regarding which methods might be called when handling the event, as well as call relations between them. In practice, this is computed using static analysis once the event's handlers are known. However, GUI events are not fired in isolation, but are part of an event sequence [7]. Each fired event contributes to improved code coverage. We take this into account when we define an event's call graph coverage:

Definition. Event call graph coverage. *Given a GUI event sequence $S = \{e_1, e_2, ..., e_n\}$, we define the call graph coverage of event e_i, with $0 < i \leq n$ as the ratio between the number of source code statements covered by S to the total number of statements from the methods in e_i's event call subgraph.*

For each event, the call graph coverage tells us how many code statements were run out of the maximum possible as determined via static analysis. As defined above, an event's code coverage can be improved if subsequent events run code from its event call graph.

However, all computed call graphs are only approximations. When computed dynamically, that is by running the target application, they might miss methods that can be called via different code paths, and are thus incomplete. When computed statically, using tools such as Soot, they might include additional edges that a more precise analysis might determine to be superfluous. This is a well-known problem in pointer analysis [8], one that is not expected to be solved for complex languages such as Java. For GUITracer, our approach was to use one of the algorithms within the Soot framework that computes an accurate call graph statically [9]. This means that while all methods that may be called are included in the call graph, it is possible that it also contains superfluous entries.

In order to build event call graphs, we required information about the event handlers installed by the target application. This was achieved using the JETracer framework that is detailed below.

B. The JETRacer Framework

JETracer is our open-source framework for real-time tracing of GUI events fired within Java applications built using one of the AWT, Swing or SWT toolkits. Available at [6], the tool consists of two modules: an *agent* and a *host* [5]. The *agent* module is GUI toolkit specific and is deployed within the target application which it instruments during start up. The process is completely transparent to the target application, and as shown within the evaluation section in [5], it does not impact its perceived performance. The JETracer agent gathers GUI event information and transmits it to the *host* module via network socket. The *host* module maintains the connection with the agent, and once an event is received it notifies any attached listeners. Adding support for new GUI toolkits or events (e.g. touch interactions) is possible by implementing a new *agent* component [5].

GUITracer is built on top of our JETracer framework, which it uses to gather event information from the target application. Figure 1 shows the architecture of our tool, including the JETracer agent and host. Code coverage capabilities are provided via the open-source JaCoCo[1] library that provides coverage information on-demand, during program execution. We harnessed this feature so that every time JETRacer sends event information it also provides updated code coverage via JaCoCo. This is done once for each GUI event fired, when handling routines have completed and control is returned to the GUI toolkit.

III. THE GUITRACER TOOL

The GUITracer [10] tool is open-sourced under the Eclipse Public License and is free to download from our website [6]. Fully implemented in Java, the application's only requirement is a Java 6 compatible platform. As a proof of concept implementation, GUITracer comes in the form of a standalone desktop application. A brief video that showcases the tool's main features is available at [10]. In order to set up an application for tracing by our tool is done by providing the location of the following artefacts using command line parameters:

- *Source code* - This folder is required in order to enable illustrating source code coverage.
- *Binaries* - The location of the binaries is required for calculating the application call graph and pre-run instrumentation.
- *Libraries* - Libraries must be provided separately in order to exclude them from the displayed callgraphs.
- *Main class* - Required to start the target application.
- *Call graph* - An optional parameter. As call graph calculation is computationally expensive, storing the call graph between runs decreases the time required to start GUITracer. Our website [6] details how to save generated call graphs in order to be reused on subsequent runs.

Figure 1 illustrates GUITracer's architecture. The tool employs our JETracer [5] framework for event and coverage trac-
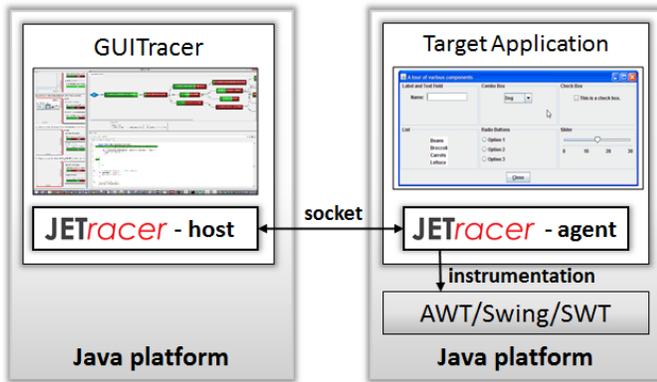
---

[1]JaCoCo - http://www.eclemma.org/jacoco/

Fig. 1. GUITracer tool architecture

ing information. Once a GUI event is fired, the JETracer agent relays the information to the framework's host component, which is integrated with GUITracer's designated handler.

Figure 2 illustrates a screenshot of the application. The target application is version 0.7.1 of the open-source FreeMind[2] mind mapping software. The tool's UI consists of three main panes - the *Event trace*, the *Call graph* and the *Source code*. We describe each of them below.

A. Event Trace Pane

GUI events fired within the target application are displayed within this pane, on the left hand side of Figure 2, which currently shows three events. The topmost event is shown with a white background as it is currently selected. The event is an *ActionEvent* fired by the toolbar button having the black outline. For each event, the tool currently displays the following information:

- *Application window screenshot*. Recorded at the time the event is fired, it allows visually identifying the event's originating widget using its black border outline. It also ensures that any visual styles active within the target application also appear in GUITracer.
- *Code coverage information*. This provides information regarding the event's code coverage. The colour coding is used consistently with most other coverage tools: green for covered and red for uncovered code, respectively. In addition, as many events might run the same code over and again, we use a lighter shade of green to illustrate code that was first run during the event's handling.

The *application line coverage* section provides information regarding how many code lines have coverage up until and including the event. This is the information most other tools also display, with the difference that in the case of GUITracer, it is gathered after each GUI event is handled. The *event line coverage* section is based on the event coverage concepts introduced within the previous section. It uses the statically computed call graph of the application and shows the event's

[2]FreeMind home - http://freemind.sourceforge.net

call graph coverage. Its purpose is to facilitate feature localization as well as provide detail regarding the link between GUI events and the source code that handles them.

As an example, after the topmost event in the trace was handled, target application coverage was of 2082 lines out of a total of 7615. The event call graph comprises 404 code lines, with a coverage of 276 lines. As shown by the light green shade, most statements handling this *ActionEvent* were not run before.

The type of GUI events displayed in the trace can be filtered. This functionality was added due to observing that in many cases, GUI applications fire a large number of events which clutter the event trace. These include focus events fired when the target application loses/gains focus as well as mouse movement events. In addition to filtering these, users can choose to hide events that do not contribute to the target application's code coverage. This is useful to hide repeated events that always take the same code path. The topmost event in the trace is selected, as shown by the white background. Once this happens, the call graph pane becomes relevant.

B. Call Graph Pane

Once an event from the trace is selected, GUITracer calculates its event call graph, and displays it in the top pane on the right hand side of Figure 2. Each call graph has exactly one *start* node, with one outgoing edge for each handler. The displayed call graph only contains application code; library as well as Java platform calls are not included, and neither are any callbacks from them. This is due to the difficulty of modelling library callbacks, which is an active topic of research [11].

The displayed call graph can be customized using the controls below the call graph panel. First of all, the graph can be displayed with method, class or package granularity. Most detailed call graphs are at method level, where each vertex represents one method. At class and package level, each vertex represents a class or package, respectively. Regardless of call graph granularity, code coverage is shown using consistent colour coding with the trace view. At class or package granularity, method calls are displayed as edges between the classes or packages they belong to, as is the case.

Changing the granularity changes both the level of detail as well as the complexity of the displayed graph. While this is application specific, method call graphs can easily contain hundreds of vertices, while class and package call graphs usually contain no more than a few dozen. In addition, users can choose to display one collated call graph, which represents all event handlers in the same pane, or have a separate tab for each handler.

These features are meant to facilitate exploration and feature location, as users can quickly retrieve information regarding the coverage of each source code entity. If a method or class-level call graph is displayed, nodes present a contextual menu allowing the element's source code to be shown in the corresponding panel.
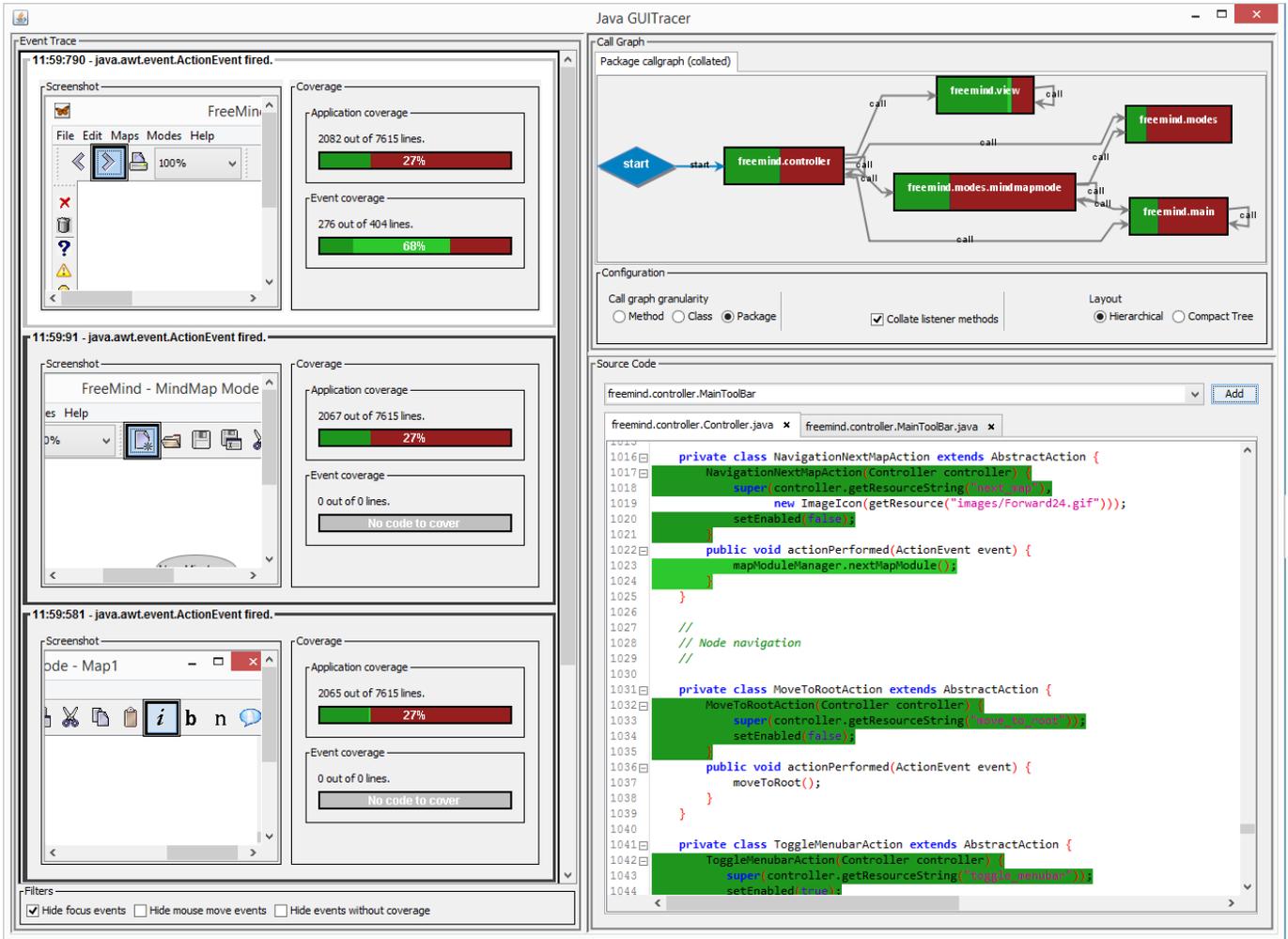
Fig. 2. GUITracer screenshot

C. Source Code Pane

This pane allows users to consult the target application's source code. Source files can be opened using the combo-box control or from the contextual menu of the call graph nodes. Source files are displayed with syntax as well as coverage highlighting. The employed colour scheme is consistent with the one previously described. The purpose of the source code pane is to illustrate that by following a top-down exploration strategy, users can start from the target application's GUI and reach the covered statements in its source code.

Given the technical challenges of implementing the GUI-Tracer tool, this proof of concept was implemented as a standalone application. However, as such tools are more useful when integrated within an IDE, the next version of the tool will be implemented as a plugin for a popular Java IDE, such as Eclipse or Netbeans. As IDE's have advanced source code editing components, the GUITracer plugin will employ them in order to showcase GUI event coverage, similar with how most unit test plugins currently work.

IV. RELATED WORK

An important body of work which GUITracer employs is the Soot framework [12]. First detailed in [13], Soot provides static analysis functions for Java programs, among which several algorithms for obtaining the static call graph [14]. Our tool uses the SPARK algorithm detailed in [9], a context insensitive algorithm that provides an adequate speed to accuracy trade-off. One of the first tools to employ Soot was JAnalyzer [15], which leveraged call graph information to provide simple graphical representation of the call relations in the target application. Our tool builds on JAnalyzer by providing complete call subgraphs of the application code starting from the entry points into event handler code.

Our tool's code coverage functionalities are inspired by efforts such as detailed by Duck et al. in [16], where a new approach for software reconnaissance based on differential code coverage is proposed. The approach is then investigated within an evaluation where users had to debug and change code in several complex GUI-driven applications. Our implementation complements the one detailed in [16] by providing context to

coverage information in the form of the GUI event trace, which facilitates identifying the relation between the source code and the user interface controls. An effort more related to GUI-driven systems is detailed in [17], where authors describe a navigation mechanism that enables source code localization for GUI elements. A controlled user study is also detailed within [17] showing important speed-ups in feature localization tasks.

One of the key challenges of implementing our tool was accurately capturing GUI event information. As this is a software tracing task, we studied previous efforts targeting Java, such as the JMonitor library developed by Karaorman and Freeman [18]. JMonitor provides event monitoring for Java by specifying *event patterns* and *event monitors*. Patterns are used to describe interesting events, and monitors act as handlers that are called once the events have taken place. The proposed library provides a generic implementation for lowest-level events such as setting the value of a class field or a method call. Another notable example is JRapture [19], a tool for capturing and replaying Java program executions by recording interactions between the program itself and the system, using accurately reproduced input sequences. Profiling can then be added to study the application during replay. JETracer differentiates itself from these tools by working on a higher abstraction level and being developed to record GUI events. This allows capturing additional information such as application screenshots as well as event listener information.

V. FUTURE WORK AND CONCLUSION

Our aim for this implementation was to provide the proof-of-concept for a tool that may find many uses during software development and maintenance. We believe the current implementation lays down the foundation for a useful tool to assist in program comprehension and feature localization for GUI-driven applications.

To the best of our knowledge, GUITracer is the first tool that successfully combines static and dynamic analyses for program comprehension of GUI-driven applications. Its creation was guided by findings from studies targeting professional developers such as those in [4], [20], which underline the fact that software exploration and comprehension are most often started at GUI level. We have also taken into account the findings within Storey et al.'s survey of software exploration tools [21] that show a lack of tools proposing a top-down approach.

In order to identify possible future improvements to GUI-Tracer, we undertook a preliminary evaluation using various versions of open-source applications such as FreeMind, jEdit and Azureus. This allowed us to discover what features are important to improve the tool's capabilities. At the present time, these include improving the visualization of large call graphs using better filtering and navigation, adding support for multi-thread programs and library callbacks [11]. A more distant issue is to provide support for unit testing by integrating our tool's visualization capabilities with well known frameworks such as JUnit.

Regarding the tool's deployment, the next step is to integrate the tool as a plugin within popular IDE's such as Eclipse and NetBeans, where GUITracer will be available while running GUI applications. At this point we plan to undertake a user-driven evaluation in order to guide further development that will make the tool as useful as possible to practitioners working on large scale GUI applications.

REFERENCES

[1] D. Hou and Y. Wang, "An empirical analysis of the evolution of user-visible features in an integrated development environment," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '09. New York, NY, USA: ACM, 2009, pp. 122–135. [Online]. Available: http://doi.acm.org/10.1145/1723028.1723044

[2] D. Hou, "Studying the evolution of the eclipse java editor," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 65–69.

[3] A. M. Memon, "A comprehensive framework for testing graphical user interfaces," Ph.D. dissertation, 2001.

[4] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014.

[5] A.-J. Molnar, "JETracer - A Framework for Java GUI Event Tracing," *Proceeding of the 10th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2015)*, pp. 207–214, 2015.

[6] GUITracer, "https://bitbucket.org/guiresearch/tools," 2015.

[7] M. E. P. Atif M. Memon, Mary L. Soffa, "Coverage criteria for GUI testing," in *Proceedings of the 8th ESEC held jointly with 9th FSE*. New York, NY, USA: ACM Press, 2001, pp. 256–267.

[8] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61.

[9] O. Lhoták, "Comparing call graphs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 37–42.

[10] GUITracer tutorial, "https://www.youtube.com/watch?v=jq2kbsehrdu," 2015.

[11] W. Zhang and B. G. Ryder, "Automatic construction of accurate application call graph with library call abstraction for java: Research articles," *Journal of Software Maintenance and Evolution*, vol. 19, no. 4, pp. 231–252, Jul. 2007.

[12] J. D. N. Arni Einarsson, "A survivor's guide to program analysis with soot," 2008.

[13] V. Sundaresan, "Practical techniques for virtual call resolution in java," McGill University, Tech. Rep., 1999.

[14] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *PASTE'01*. ACM Press, 2001, pp. 54–61.

[15] E. Bodden, "Janalyzer: A visual static analyzer for java," 2003, as contribution for the SET Awards 2003, category computing.

[16] K. Duck and G. Murphy, "Reducing code navigation effort with differential code coverage," 2008. [Online]. Available: http://www.cs.ubc.ca/cgi-bin/tr/2008/TR-2008-14

[17] A. L. Santos, "GUI code tracing through direct program interaction," *Journal of Software: Evolution and Process*, vol. 26, no. 6, pp. 552–568, 2014.

[18] M. Karaorman and J. Freeman, "jMonitor: Java Runtime Event Specification and Monitoring Library," *Electron. Notes Theor. Comput. Sci.*, vol. 113, pp. 181–200, Jan. 2005.

[19] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay Tool for Observation-based Testing," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 158–167, Aug. 2000.

[20] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[21] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *J. Syst. Softw.*, vol. 44, no. 3, pp. 171–185, Jan. 1999.