Visualization and Testing of GUI Applications

Arthur-Jozsef Molnar

Departament of Computer Science

Babeş-Bolyai University of Cluj-Napoca

- Thesis Summary -

The author was supported by programs co-financed by The Sectoral Operational Programme Human Resources Development, Contract POS DRU 6/1.5/S/3 - "Doctoral studies: through science towards society"

2012 February

Contents

Publications related to this paper

Molnar, A.J.

jSET - Java Software Evolution Tracker

Presented during *KEPT 2011 Conference, Cluj-Napoca*

Extended abstract in *Studia Informatica UBB*, Volume *LVI* Number *3*, pages *15–20* **(BDI)**

Full paper available in volume Proceedings of KEPT-2011 with *Cluj University Press*, under *ISSN 2067-1180*, pages 259–270. **(ISI Proceedings)**

Molnar, A.J.

A Heuristic Process for GUI Widget Matching Across Application Versions

Abstract to be presented during *MaCS 2012 Conference, Siofok, Hungary*

Full paper accepted for publication in *Annales Universitatis. Scientiarum Budapestinensis, Sectio Computatorica* **(BDI)**

Molnar, A.J.

A Software Repository and Toolset for Empirical Software Research

To be submitted for publication in *Studia Informatica UBB*, *ISSN 1224-869x*, Cluj-Napoca **(BDI)**

Molnar, A.J.

An Initial Study on GUI Test Case Replayability

To be submitted for publication at *IEEE International Conference on Automation, Quality and Testing, Robotics AQTR 2012, Cluj-Napoca* **(ISI Proceedings)**

Introduction

This thesis is the result of my original research on the topics of GUI driven application visualization and testing. My work started in 2008 under the supervision of Prof. dr. Bazil Pârv.

My work was focused on two main avenues of research: improve the state of GUI software visualization and analysis by incorporating latest tools developed within the academia and provide new means for testing GUI driven applications.

Software visualization is the representation of information gained from studying software systems. Our research consists of harnessing state of the art analysis tools developed in academia to provide valuable information about the development of the target software system. We harness the Soot[1] static analyzer and use it as the input of algorithms that allow tracking the evolution of target GUI applications (7).

Our main research direction regards the automation of GUI-driven application testing. Most applications today use GUIs to interact with users, so their correctness has never been of more concern. Our work in GUI testing is model based and is linked with the research efforts undertaken by a team from Maryland University lead by prof. Atif Memon[2].

This thesis is divided into five Chapters as follows.

Chapter 1 introduces required preliminaries for our research. We briefly survey relevant aspects of automating the testing process and introduce work related to our research. In **Chapter 2** we introduce an extensible software repository built to enable empirical software research. **Chapter 3** is dedicated to our original research regarding an extensible framework of software components to enable software visualization and testing. **Chapter 4** contains our original contributions to improving the state of the art in GUI test case maintenance. We implement an extensible heuristic process able to achieve high accuracy when matching functionally equivalent GUI elements and perform an extensive case study to assess its performance. The final part of our work is **Chapter 5**, where our efforts in software visualization and GUI test case maintenance are united toward the common goal of improving the state of the art in GUI testing.

Our original contributions are contained in Chapters 2,3,4 and 5 and are as follows:

[1] Soot homepage - http://www.sable.mcgill.ca/soot/
[2] http://www.cs.umd.edu/ atif/

- A set of criteria usable when choosing target applications for empirical software research.
- A *Project* system used for storing artifacts related to a software system's state.
- Supportive tooling that allows easy programmatic access to Project instances.
- A software repository consisting of 30 versions of two popular, complex, GUI driven open-source applications.
- Script files that automate obtaining relevant project artifacts whenever desired.
- An extensible framework to support the creation and assembly of advanced software tools.
- Three implemented software components, or *View*s that provide features reusable in both software development processes and in empirical software research. We reuse these components in Chapters 4 and 5 when we instantiate new software tools based on the presented framework.
- The *jSET - Java Software Evolution Tracker* tool, obtained by assembling already developed views into a software visualization and analysis tool that provides the means to examine the evolution of a software system from a top to bottom perspective, starting with changes in the graphical user interface all the way to source code modifications.
- A three step heuristic process based on a prioritized list of heuristics able to match GUI elements across different versions of a target application.
- Two execution strategy algorithms that control the configurable steps of the proposed process.
- Several heuristic and heuristic-factory algorithms able to achieve high accuracy in matching GUI application windows and widgets.
- An extension to our software repository that contains correctly matched GUI elements for 28 version pairs of the FreeMind and jEdit applications usable in further research.

- A number of metrics generally usable in assessing the accuracy of a widget matching process.
- A highly accurate heuristic configuration usable as the base of a long-term GUI test case maintenance system.
- An extensive case study that examines the accuracy and feasibility of our best heuristic configuration.
- An analysis of classes of typical heuristic mistakes.
- A theoretical foundation for combining code coverage metrics with static analysis tools in the context of GUI driven applications.
- Two software components that implement necessary functionality to provide information about GUI test step and test case coverage.
- A software tool that allows a detailed examination of the execution of GUI test cases on the target application.
- An extensive case study employed to provide answers related to questions about GUI test case replayability in the context of evolving applications and the feasibility of our heuristic process in enabling long-term GUI test case maintenance.
- A proposed automated regression testing process for GUI applications that combines our research from Chapters 2,3,4 and 5 into a unitary whole that is ready to be deployed within the industry.

1

Introduction

This Chapter serves as an introduction to our original work. It is the result of our extensive overview of existing literature on software visualization and GUI application testing. Section 1.1.1 details some of the general aspects of test automation, while Subchapter 1.1.2 is dedicated specifically to GUI application testing. A brief overview of available tooling for testing GUI applications is provided in Sections 1.1.2.1 and 1.1.2.2. We reserve Subchapter 1.2 to presenting the Soot and GUITAR frameworks in detail, as we extensively use them in the course of our research.

1.1 Testing GUI Applications

This Subchapter surveys existing approaches in software testing. In Section 1.1.1 we survey some of the most important aspects regarding testing and present state of the art contributions we believe are most relevant for our work, while Section 1.1.2 is dedicated to introducing existing approaches in GUI-driven application testing.

1.1.1 Approaches in Software Testing

The growing importance of software testing was identified as early as 1975 in two pivotal papers: Goodenough provided the fundamental theorem of testing and an early taxonomy for program errors in (15), where he presents a number of example programs for analysis. Then he uses them to draw conclusions about how to develop effective program tests and how to derive test cases using program specifications. The following sections describe some aspects of software testing decisive for carrying out our research.

We provide information about some of today's paradigms and methodologies in testing and refer to them in the following Chapters of our work, where we employ them or study the feasibility of using them in further efforts.

1.1.1.1 Automating the Testing Process

Bach provides an interesting insight into late '90's automated testing tools in (1), where some of the assumptions on test automation are debunked and a *sensible approach* to automated testing is provided. The author's main points regard choosing suitable tools and combining manual and automated approaches rather than using one to replace the other. The result of more in-depth work is presented by Ramler and Wolfmaier, who use an economic approach for test automation in (41). A more recent report by Berner et al. mirrors the conclusions from (1) regarding difficulties encountered by production software with maintaining testable design and successfully combining manual and automated techniques.

The course of our research was guided by these findings. We generally try to steer clear of an "automate all" approach and provide means of automating some aspects of testing which we believe can be completely relegated to the computer. Additionally, we provide new capabilities on software visualization which improve manual testing by providing cross-version application change and static analysis result visualization.

1.1.1.2 The Test Oracle Problem

No discussion on test automation is complete without touching the test oracle problem (43). Somewhat like its historical counterpart, the test oracle provides the means for checking whether the system under test has behaved correctly on a particular execution (2) by providing the correct "answer", or result of the test run.

We posit that no process of testing automation is viable without providing a solution to the oracle problem. An automated oracle is a specialized software system that when given the input and output of a test run will provide a *pass/fail* rating on the output of the executed test without human intervention (43). Staats et al. discuss important automated oracle properties (43) such as *completeness* (does the oracle conform to program specification), *soundness* (is the oracle result correct) and *perfection* (if the oracle is both complete and sound).

An early effort to provide an automated oracle in the field of GUI testing was undertaken by Memon, Pollack and Soffa in (34). The state of the user interface is formally modeled using the theoretical aspects from (30) and its expected state can be derived at any time using the initial state of the GUI and the sequence of actions applied. We find a variation of this procedure in (27), where a golden standard is used for extracting the GUI state after each action in order to obtain oracle information. A further study of automated oracles in found in (55), where Xie and Memon use several levels of oracle information and oracle procedure (27) in order to determine trade-offs between accuracy and cost of testing.

1.1.1.3 Test Suite Evaluation

Proposed solutions to the assessment of test suite adequacy lay in using methods such as fault injection and mutation testing. Fault injection consists of purposefully altering a program so that a fault is introduced. To apply mutation testing, one starts from the original program from which mutants are derived by injecting faults. The "strength" of a test suite is measured by running it on all the mutants and checking how many of them are detected by test cases, referred in literature as *killing* the mutant. This allows an approximation of the number of undiscovered errors in the program by extrapolating how many of the introduced mutants were killed.

Mutation testing techniques are employed in many GUI testing case studies. In (55), Xie and Memon use fault seeding to evaluate the accuracy of GUI oracles. In (44), Strecker and Memon use it to create a large number of single-mutation mutants to evaluate the impact of testing suites on fault detection. Fault seeding is used again in (58), where authors employ it to study the effectiveness of the described mode-driven test generation technique.

1.1.2 State of the Art in GUI Application Testing

As user interaction with GUI applications happens through the graphical interface, its correct functioning is considered crucial (11). A study concerning customer reported GUI defects in complex applications was undertaken by Robinson and Brooks in (42), targeting *"two industrial software systems developed at ABB" (42)*. From their results they concluded that:

1. "65% of the defects resulted in a loss of some functionality to the customer.
2. 60% of the defects found were in the GUI, as opposed to the application itself.
3. defects in the GUI took longer to fix than defect in the application itself" (From (42)).

As outlined in many studies (11, 42) testing GUI applications is not trivial as GUI related code can comprise up to half of all application code (30). As pointed out in (11), there are many situations where application errors are reflected in the application GUI. This is backed by Xie and Memon's case study *67* where several versions of highly popular open source applications are GUI tested and previously unknown errors are uncovered. We posit that improvements in testing GUI driven applications brings added value to software products by shortening the time required to discover bugs and implement necessary fixes.

1.1.2.1 Tools Based on Capture and Replay

Capture-replay tools (17) work in the two phases that spawned their name. During the first capture phase the application records the tester's interaction with the target application and persists it for later use. The recorded data is reused during the automated replay phase, where the test case containing the recorded interaction is replayed on the target application. This approach however presents several difficulties:

- Intended behaviour of the system under test can change over time which might trigger the testing software to mark the new behaviour as faulty during replay.
- As GUI software evolves changes occur in the GUI structure which makes recorded interactions useless as the components used when test cases were captured have been changed or moved and are not recognized by the replay component.
- This approach solves at best only half the problem as test case creation and recording is still a completely manual undertaking.

Due to these inherent limitations, modern approaches usually combine capture/replay tools with model-based techniques, like the GUI Test Generator described by Nguyen et al. in (39), or Bertolini and Mota's approach in (3), where a framework for GUI test case design using a controlled natural language (13) is proposed for test case generation.

1.1.2.2 Model Based Tools

The most important body of research in the theory and practice of model-based GUI testing comes from the University of Maryland, where the research group led by prof. Memon[1] advanced the state of the art in both theoretical and practical aspects of the testing process.

The first paper that must be mentioned is Memon's PhD thesis (30), defended in 2001, which provides some important theoretical concepts that were used in future work, such as a definition for the class of GUIs targeted by the research, a definition for the state of a GUI and a definition for the *Event Flow Graph* (30) which models the valid flow of GUI events within the application.

The work of the Maryland research group is important as we use the provided theory and developed toolset (detailed in Section 1.2.2) in the original research presented in this paper. The first important contribution regards a definition for the class of graphical user interfaces under research, which Memon defines as follows:

Definition 1.1 *A Graphical User Interface is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI (From (30))*

Based on the definition provided above, Memon then defines (30) the state of the GUI at a moment in time in terms of:

- *its objects* $\mathrm{O} = o_1, o_2, \ldots, o_m$, *and*
- *the properties* $\mathrm{P} = p_1, p_2, \ldots, p_n$ *of those objects. Each property* p_i *is an* n_i*-ary Boolean relation, for* $n_i \geq 1$ *where the first argument is an object* o_1 *included in* O. *If* $n_i > 1$, *the last argument may either be an object or a property value, and all intermediate arguments are objects. The (optional) property value is a constant drawn from a set associated with the property in question: for instance, the property* background-color *has an associated set of values* (white, yellow, pink, etc). *One of its properties might be called* Caption *with a value of "Cancel".*

[1] University of Maryland - Event Driven Software Lab - http://www.cs.umd.edu/~atif/edsl

Therefore, Memon defines the state of the GUI at a particular time t *as the set* P *of all the properties of all the objects* O *that the GUI contains (From (30)).*

On these basis, Memon introduces (30) the *event-flow graph*, which given a component C is defined as follows:

Definition 1.2 *The event-flow graph is a 4-tuple* $< V, E, B, I >$ *where:*

1. V is a set of vertices representing all the events in the component. Each v included in V represents an event in C.
2. E included in $V \times V$ is a set of directed edges between vertices. Event e_i follows e_j iif e_j may be performed immediately after e_i. An edge (v_x, v_y) is included in E *iif* the event represented by v_y follows the event represented by v_x
3. B included in V is a set of vertices representing those events of C that are available to the user when the component is first invoked.
4. I included in V is the set of restricted-focus events of the component. (From (30))

The importance of this paper, in the light of the research following it is that it provides a formal specification for a class of GUIs and models structures that are required in their representation and testing. The theoretical part detailed above was used in the implementation of the GUITAR GUI Testing Framework (26, 47), which we describe in detail in Section 1.2.2 and proceed to use in our research detailed in Chapters 2 throughout 5.

The theoretical part of (30) has seen wide use in empirical software research and many papers (21, 27, 28, 29, 32, 34, 53, 54, 55, 56) that detail improvements to GUI testing processes employ it.

1.2 Advanced Research Frameworks

The following sections present two research frameworks that provide the backbone of our research. The Soot framework provides static analysis features for the Java platform, making it an extremely useful tool when a white-box approach is considered. Meanwhile, the GUITAR framework only uses on-screen information and thus can be classified as a black-box tool.

1.2.1 The Soot Static Analysis Framework

This section introduces Soot (48), a research framework for static analysis[1] of Java programs developed at the McGill University. Currently at version 2.4.0[2], it has a long history with many new features added during its development. Its website (49) mentions a long list of users from the academic environment who use it both as course material and for research purposes.

The Soot framework provides implementations for performing *class hierarchy analysis* - CHA, first described as a compiler optimization technique in (14). The CHA analysis provides information about the types of instances receiving messages and enables whole-program analysis and optimization by computing the program call graph as described in (46).

The program call graph is a statically computed directed graph that has the program's methods as nodes and where each directed edge represents a "calls" relationship between methods. Since it is statically generated, there is no order between a node's (method's) outgoing edges as we cannot know in what order the methods will be called when the program is executed. As CHA analysis is the cheapest with regards to time and memory, it gives a significant over-approximation of the call graph, which leads to the problem of developing more advanced algorithms to eliminate redundancies (46). Such algorithms are implemented in Soot and described in Sundaresan's Master's Thesis (45). He provides the *rapid type analysis* - RTA algorithm which provides precise analyses by taking into account that call receiver instances must belong to instantiated types (45). An important contribution to Soot was made by Ondrej Lhotak, the developer of the SPARK points-to analysis framework integrated in Soot (23).

Our main interest in Soot is due to its callgraph building capabilities which provide precise call graph representation for complex software, as shown using case studies undertaken in (23, 24, 45). We believe that by harnessing these capabilities we can provide advanced tools for software visualization and analysis.

1.2.2 The GUITAR GUI Testing Framework

The implementation of the GUITAR testing framework was a major step in advancing the state of the art in GUI-driven application testing. Its purpose was to allow au-

[1] Static in the sense that the analyzed program is not started, as described in (18), section 4.5.
[2] As of December, 2011

tomation of testing processes by including tools capable of generating, executing and evaluating test cases. GUITAR (47) has four major components, presented here in the order they are usually employed:

- *GUIRipper* is usually the first tool to be used when deploying GUITAR. Described in detail by Memon et al. in (26), it can run the target software and save its GUI model to an XML data structure. We heavily employ the GUIRipper tool's Java implementation in our research to obtain GUI models for the applications in the repository described in the 2nd Chapter.

- *GUI2EFG*. This tool takes the model obtained using the ripper and creates the event-flow-graph (or EFG) of the application. The EFG is important as it details valid sequences of events for the target GUI, enabling the creation of executable GUI test cases (31).

- *TestCaseGenerator*. The Test Case Generator's input is the XML file representing the event-flow-graph obtained in the previous step. Its output is a test suite represented by a collection of test cases which are generated by plugins (16).

- *TestReplayer*. The Test Replayer runs the test cases generated by starting the application and firing the test case events in correct order. (12, 28).

The original research detailed in this paper uses the GUITAR framework for recording target application GUI models and for an empirical assessment of our widget matching work on GUI test case maintenance.

2

An Application Repository for Empirical Research

This chapter presents original work in building an extensible software repository consisting of several versions of popular open-source applications chosen to serve as experimental and case study targets in our research. Subchapter 2.1 presents the criteria we used when searching for complex applications eligible for use in empirical software engineering while Subchapter 2.2 consists of original work describing our repository data model and associated software tooling. In Subchapter 2.3 we describe two popular open-source software applications that provide the bulk of our repository data. Finally, Subchapter 2.4 summarizes our efforts and proposes directions to extend our repository.

Our original contributions are listed in the opening Chapter of this work and have been submitted for publication (10).

2.1 Criteria for Suitable Applications

The current Section discusses some of the important characteristics software applications should have to qualify as target applications for empirical research. While the presented criteria were arrived at with consideration to our particular needs we believe them to be generally applicable and to provide a good foundation when selecting experimental or case study targets:

- *Usability*: We believe all meaningful research must target useful software. We believe a vibrant user and developer community that share ideas and actively guide the development of the application to be a good sign of such software.

- *Complexity*: Real software is complex and many methodologies and metrics are available for assessing complexity. Thorough research should be based on several relevant complexity metrics and should be tied with the usability criteria mentioned above.

- *Authorship*: We believe that by picking applications totally unrelated to the research effort goes a long way in proving its general applicability. The best way to achieve this is to select target data produced by third parties uninterested and unrelated to the current research efforts.

- *Availability*: Our research aims to improve the state of the art in processes that are part of software development. In order to validate our ideas we need access to the software we validate our research on. This has multiple implications in both legal and technical directions, as target applications should be freely available and changeable while being both easy to set up, use and tear down when no longer needed.

- *Simplicity*: The need to access multiple versions of the same software application raises the importance of the simplicity criterion. In order to have multiple programs set up, configured and ready to run we searched for applications that do not require complex 3rd party packages that are challenging to set up, configure and clean up.

2.2 A Proposed Repository Model

Our repository is modeled as a set of *projects*. Each project represents a target application at a given moment in time. Having multiple projects that capture the same application helps when studying regression testing, tracking and analyzing software evolution and so on. Each project has associated the following artifacts:

- *Project File*. This is an XML file that contains the project's name and location of the artifacts that comprise the project.

- *Application binaries.* Each project contains two directories: one that holds the compiled application data and the other for required libraries. Each project also contains a script that can be used to start the application.
- *Application sources.* Each project has a source directory that contains the application source code.
- *GUI File.* Contains the output of running GUITAR's GUIRipper on the target application. This provides the application's GUI model in XML format.
- *Widget Screenshots.* We modified the GUIRipper application to enable recording application screenshots that are associated with the ripped widgets. This allows studying how the GUI widgets actually look like without having to start the application.
- *Application callgraph.* This is the callgraph obtained by running Soot's SPARK implementation over the target application.

Our model's most pressing limitation concerns the callgraph itself. As the Soot framework only works on Java programs, target applications implemented using other platforms will not have their callgraph recorded. This is due to the complex nature of computing the callgraph; currently we do not have knowledge of tools comparable to Soot that work on other platforms.

Our repository is structured so that every project has its own SVN directory, making it easy to select and download individual projects. In addition to providing actual repository data, our toolset allows programmatic access to target application data. Each project is programmatically represented by an instance of the *jset.project.Project* class which provides access to the project artifacts discussed above. Loading *Project* instances is done via the *jset.project.ProjectService* class that provides the required methods.

2.3 Repository Contents

The search based on the criteria laid out in the first Subchapter of the present Chapter led us to two target applications: the FreeMind (50) mindmapping software and the jEdit (51) text editor. Both of them are available on the SourceForge website and are

provided with free-software licenses that allow altering and distributing their source code. The following Sections discuss both these applications.

2.3.1 FreeMind

The FreeMind mind mapping software is developed on the Java platform and is available under GNU's GPL license. Our repository contains 13 distinct versions of the FreeMind application, dated between November 2000 and September 2007. Table 2.1 contains details regarding the downloaded versions such as CVS time, approximate corresponding release version and the number of classes, lines of code and GUI widgets.

Version	CVS Timestamp	Classes	LOC	Widgets	Windows
0.1.0	01.11.2000	77	3597	101	1
0.2.0	01.12.2000	90	4101	106	1
0.2.0	01.01.2001	106	4453	132	1
0.3.1	01.04.2001	117	6608	127	1
0.3.1	01.05.2001	121	7255	134	1
0.3.1	01.06.2001	126	7502	136	1
0.3.1	01.07.2001	127	7698	137	1
0.4.0	01.08.2001	127	7708	137	1
0.6.7	01.12.2003	175	11981	244	1
0.6.7	01.01.2004	180	12302	251	1
0.6.7	01.02.2004	182	12619	251	1
0.6.7	01.03.2004	182	12651	251	1
0.8.0	01.09.2007	544	65616	280	1

Table 2.1: Versions of FreeMind used

2.3.2 jEdit

The jEdit application is a text editor written using Java and similar to FreeMind, is available under the GNU GPL license. For building our application repository we used a number of 17 versions of this application. Similar to our approach with FreeMind, we only selected distinct versions that had at least one month of development between them. The first version of jEdit considered is the 2.3pre2 version available since January 29th, 2000, while the latest version we used is 4.3.2final, released on May 10th, 2010. Table 2.2 presents the versions in our repository together with some key information related to each version, as in the case of the FreeMind application.

Version	CVS Timestamp	Classes	LOC	Widgets	Windows
2.3pre2	29.01.2000	332	23709	482	12
2.3final	11.03.2000	347	25260	533	14
2.4final	23.04.2000	357	25951	559	14
2.5pre5	05.06.2000	416	30949	699	16
2.5final	08.07.2000	418	31085	701	16
2.6pre7	23.09.2000	456	35020	591	12
2.6final	04.11.2000	458	35544	600	12
3.0final	25.12.2000	352	44712	584	13
3.1pre1	10.02.2001	361	45958	590	13
3.1pre3	11.03.2001	361	46165	596	13
3.1final	22.04.2001	373	47136	648	13
3.2final	29.08.2001	430	53735	666	12
4.0final	12.04.2002	504	61918	736	13
4.2pre2	30.05.2003	612	72759	772	13
4.2final	01.12.2004	650	81755	860	14
4.3.0final	23.12.2009	872	106398	992	16
4.3.2final	10.05.2010	872	106510	992	16

Table 2.2: Versions of jEdit used

2.4 Conclusions and Further Work

This chapter describes our efforts in implementing an extensible repository of software applications suitable as targets of empirical research in many fields of software engineering such as application visualization, code analysis and software testing. We believe the present repository is especially well suited for research regarding black and white-box software, GUI testing and regression testing.

3

An Extensible Framework for GUI Visualization and Testing

This chapter presents our research in developing an extensible software framework and associated components that provide advanced software analysis and visualization capabilities. Save for Subchapter **??** which details required preliminaries and related work this chapter is entirely original.

Subchapter 3.1 describes our rationale for developing the present tooling, while Subchapter 3.2 provide a high level overview of the implementation. Subchapter 3.3 details three reusable components implemented to aid our research and Subchapter 3.4 introduces our original jSET tool obtained by assembling the implemented components. Known limitations of jSET are explored in Subchapter 3.4.3, while conclusions and directions of future work are outlined in Subchapter 3.5.

Our original contributions presented in the introductory chapter and have been presented at the KEPT 2011 conference (7) and are available in extended form in (6).

3.1 Introduction

By studying the evolution of widely used IDE's such as Eclipse (19, 20) one can see that each new version ships with better and more complex tools for aiding professionals in building higher quality software faster. However a look at today's software tools reveals that while most provide some visualization and analysis support there is a clear lack of tools that harness complex algorithms to provide unified program visualisation and

analysis across application layers for GUI driven software. To address this issue we developed our supporting framework using the Java platform in modular fashion to enable assembling available components into different software tools, or configurations, in order to provide new features.

3.2 Extensible Design

The basic building block of our supportive framework is the *Component*, which is a software component that fuses behaviour and graphical interaction layer to provide advanced functionality. New implementations must extend the *AbstractView* class which provides some basic functionality and a property-map that is used as the component's context. Our framework's instantiations consist of multiple such components that when combined, provide the complex functionality required. The following sections of this Subchapter detail some of the existing components, while additional implementations are detailed in Subchapters **??** and 5.1.

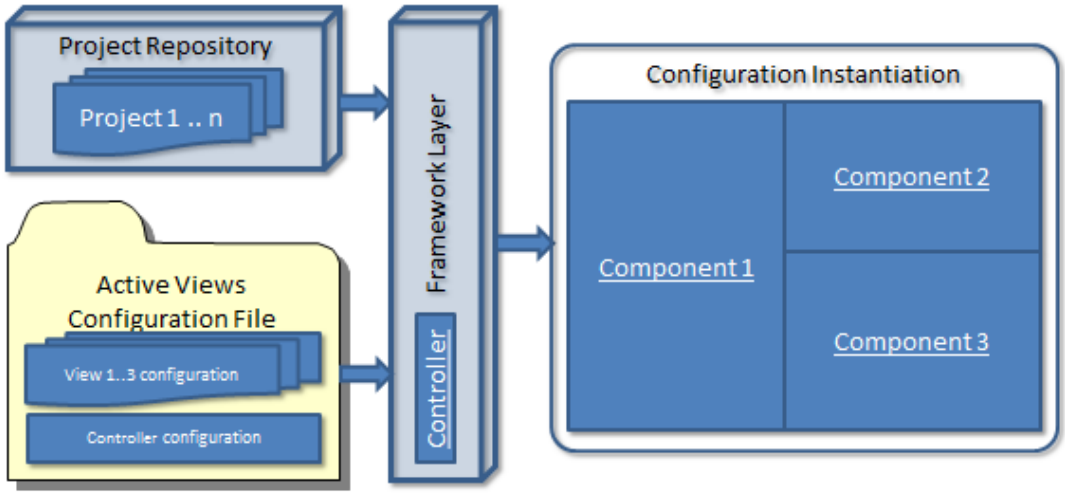

Figure 3.1: Framework Deployment

Tools implemented using our framework are instantiated using information provided in an XML input file, modeled as the yellow dossier. The most important data regards the *Controller* to be used, as it is responsible with linking components and providing unitary behaviour. When the tool is started, the framework code, shown in Figure 3.1 instantiates the supplied controller and necessary components which are then laid out.

Input data for the tool is in the form of *Projects*, as described in Subchapter 2.2 in the 2nd Chapter.

The first of our implemented tools is jSET, detailed in the following Subchapter while Section 5.1.4 presents the *GUI Test Suite Manager* tool which enables management of GUI test suites.

3.3 Implemented Components

This section describes the components that comprise the jSET application, all of which were reused as part of other tool configurations in our research.

3.3.1 GUI Compare View

This component was implemented in order to provide a visual representation of target GUIs. It uses a tree structure where the root node's direct children are GUI windows, while widgets start from the third level of the hierarchy. A popular open-source application using a similar implementation is SwingExplorer[1]. However, there are certain important differences which make the *GUI Compare View* component more complex and providing more functionality than other, standard implementations.

The main differentiator is our component's ability to integrate two GUI versions of the same application into a single display area. For example, Figure 3.2 shows a screen capture of this view displaying GUIs of jEdit versions 2.3pre2 and 2.3 final. The GUI models displayed are taken from *Project* instances using our own GUI model, described in detail in the 2nd Chapter of this work. This hierarchy is computed by comparing the hierarchies of the loaded projects; GUI elements are matched by their extracted properties. Note that some of the GUI items are color coded. Red items represent GUI elements that can no longer be found on the newer version, while items in blue are elements that are not found on the older version. Green is used to draw those elements that are present in both loaded versions but were affected by underlying code changes between them.

[1] http://www.swingexplorer.com

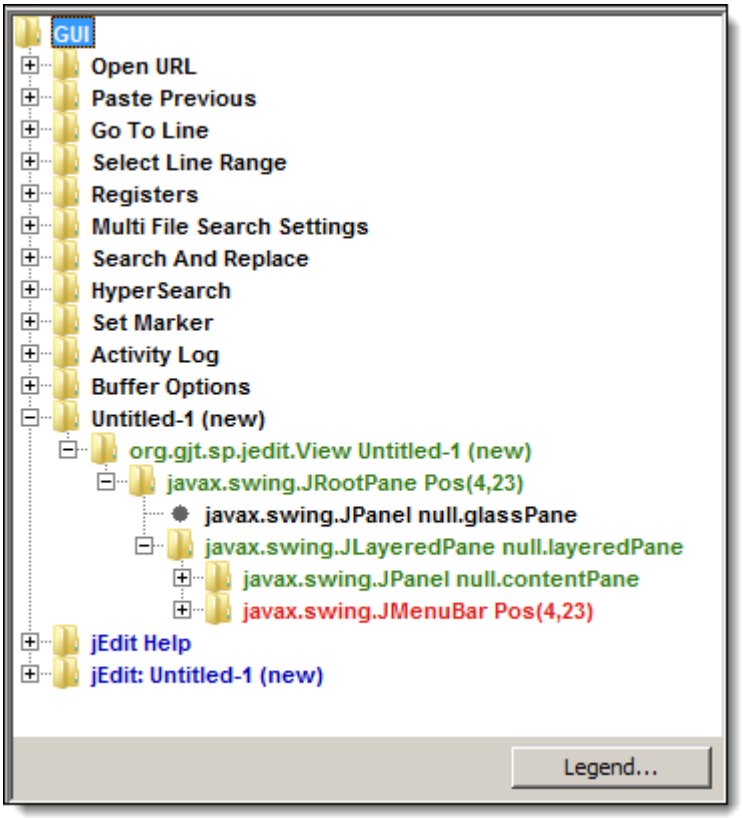

Figure 3.2: GUI Compare View

3.3.2 Widget Information View

This is the most basic, yet important of the implemented components. It is used to provide information about GUI elements using the loaded project model. The current implementation will display three tabs: the first can be used to visually locate the elements on the GUI window they belong to, while the second and third display the GUI element's properties and attached listeners, if any.

In Figure 3.3 the view is used to display two equivalent menu items of FreeMind application versions. Like in the case of the GUI Compare View, similar implementations are available using open-source software such as SwingExplorer or IDE's with GUI building capabilities such as NetBeans' Swing GUI Builder[1] or the Eclipse Window Builder[2].

[1] http://netbeans.org/features/java/swing.html
[2] http://www.eclipse.org/windowbuilder

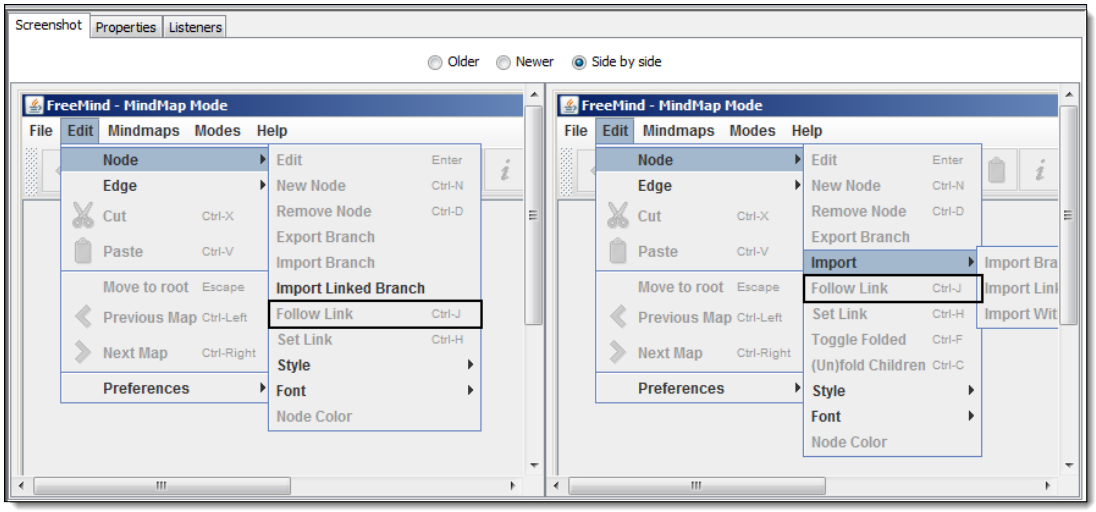

Figure 3.3: Widget Information View

3.3.3 Call Graph View

This view is one of our most important original contributions presented in this chapter. The *Call Graph View* component provides the necessary functionality for visualizing application callgraphs. As discussed in the 1st Chapter, a Java application callgraph usually consists of tens of thousands of nodes, most of which belong to framework code, therefore fully displaying it would occupy too much screen real estate and the result would not be easily browsable. To alleviate this problem, our implementation collates all framework and library methods into single nodes that are labeled as *"Framework"*, as shown in Figure 3.4.

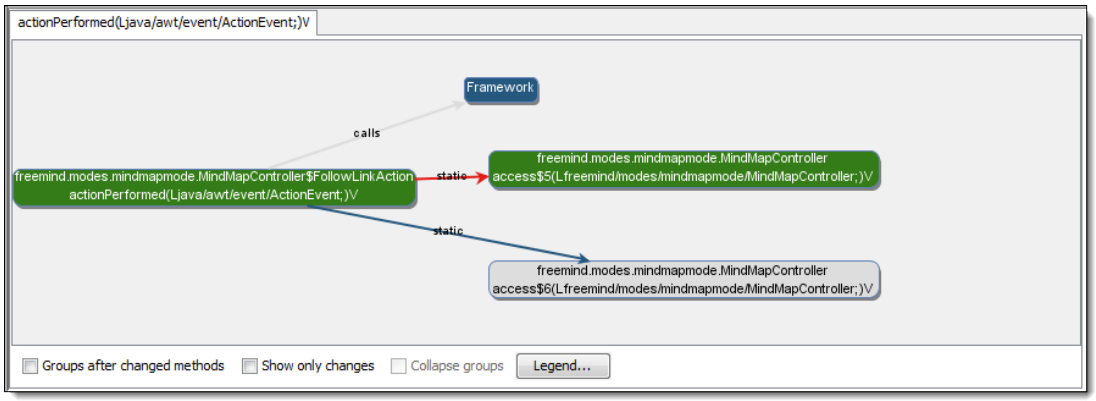

Figure 3.4: Call Graph View

Figure 3.4 shows the collated call subgraphs starting from the application event handlers of the menu items displayed in Figure 3.3 and bounded by framework methods. Application methods are represented by nodes labeled using their signature, while directed edges are used to represent calling relations.

As the subgraph in Figure 3.4 is a collation of two subgraphs from different application versions[1], color coding is used to supply additional information. Figure 3.4 provides an adequate example of the colors used, which are as follows:

- *Light-gray* is used for methods or method calls that have not changed between the studied versions.
- *Green* is used for displaying methods with bytecode changes.
- *Red* is used for methods and calls that have been deleted in the new version.
- *Blue* is used for new methods and calls.

Our desire is to show as much information as possible displaying the least amount of data. To accomplish this, our approach is to provide means to abstract some of the data, which can be done using the toolbar at the bottom of the graph display, also shown in Figure 3.4. The toolbar contains controls that allow unchanged method nodes to be collapsed in subgraphs, leaving only methods that were changed in plain view.

3.4 jSET - Java Software Evolution Tracker

This tool is the first result of implementing our supportive framework. jSET is a visualization and analysis tool that provides practitioners the means to examine the evolution of a software system from a top to bottom perspective, starting with changes in the graphical user interface all the way to source code modifications.

The jSET tool is built using the three components discussed above and uses input data obtained from the *Project* model detailed in Subchapter 2.2. The jSET tool can work in exploration or comparison mode, as detailed in the following sections.

[1] FreeMind CVS snapshots from November and December 2000

3.4.1 Project Exploration

This mode is started by providing a single *Project* instance when starting jSET. Figure 3.5 shows the application in exploration mode analyzing a January 2001 CVS snapshot of FreeMind.

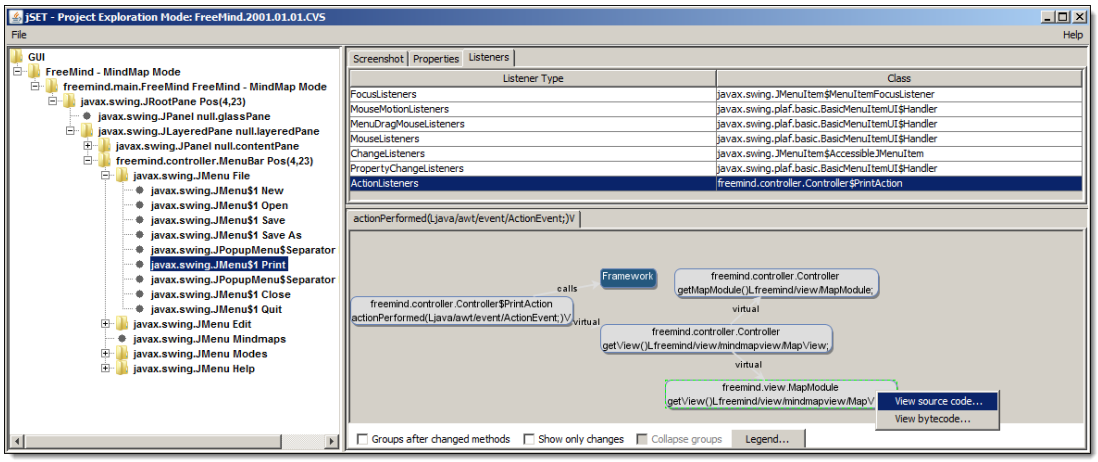

Figure 3.5: jSET in Exploration Mode

The screen capture in Figure 3.5 reveals the configuration of our implemented components for the jSET tool: the left side of the screen displays the target application GUI hierarchy using the *GUI Compare View*, while the right side of the screen contains widget and callgraph information provided using our *Widget Information* and *Callgraph View* components.

These independent components are coordinated by our *jSETController*, which is the heart of the tool. When starting this mode, the GUI hierarchy will be displayed on the left side (note that all nodes are in black, as there is no comparison being performed). When the user selects one of the GUI nodes, relevant information appears in the *Widget Information View* which allows visually identifying the selected element on its parent window. Figure 3.5 shows the third tab of the view which is used to display event handlers associated with the selected GUI element. Note the handlers in the *javax.swing.** packages that are automatically managed by the platform. In addition, we see one application handler, clicking which provides us with a section of the application callgraph which starts at the event handler's method(s) and comprises all called application methods, excluding library and framework calls.

The main original contributions implemented regard the possibility of studying an application snapshot across all application layers, regardless of how they were implemented. The GUI is browsable and its elements are easily recognized as they are highlighted on application screenshots.

3.4.2 Project Comparison

Our most important contribution to software tooling presented in the current Chapter is jSET's comparison mode. By selecting two target application snapshots, jSET will start in this mode and will provide information about application changes across multiple layers. Changes in the GUI are revealed by the GUI Compare View, while element property changes are shown in the tabs of the Widget Information View. More over, changes to the application callgraph are revealed by the Callgraph View component described in Subchapter 3.3.3.

In addition to already described functionalities, the comparison mode allows comparing bytecode and source code for displayed methods by using the context menu available for nodes representing application methods. Testers can use the comparison mode to determine what areas of the GUI are newly implemented or have been recently changed and adjust testing plans accordingly. The tool also enables users to easily assess the magnitude of changes across versions and on a broader scale to track the evolution of the target application across multiple versions.

3.4.3 Limitations

Some of the more pressing limitations of the jSET tool are represented by dynamic user interfaces, widgets interacting using library callback, native method handling and the use of reflection in the analyzed program. Our aim for future versions is provide users with options to control or to eliminate most of these issues.

3.5 Conclusion and Future Work

The framework and tools described in these pages is used throughout our research. Additional view and configuration implementations are described in future chapters where they are of most relevance. Among possible improvements, we note adding

support for tracking changes across multiple versions and integrating our framework into well known IDEs as plugins.

4

A Heuristic Process for GUI Widget Matching Across Application Versions

This chapter details our research regarding a high accuracy heuristic process that enables long term GUI test case maintenance and cross-version GUI visualization and analysis. Save for the first Subchapter which introduces required preliminaries, this chapter presents entirely original work.

This Chapter is structured as follows. Subchapter 4.1 introduces required preliminaries to our work and in Subchapter 4.2 we present an original heuristic-based process for carrying out GUI element matching. Subchapter 4.3 details our implementations able to achieve high accuracy when matching GUI elements. Our original developments are put to the test in Subchapter 4.5, where we describe an extensive case study used to assess the accuracy of the proposed process on the applications in our software repository. Further on, typical classes of errors are analyzed and used in finding new research directions for improving the process.

Our contributions presented in this chapter are listed in the introductory Chapter of this work and will be presented during MaCS 2012 (5), with an extended versions of the work accepted for further publication (8)

4.1 Preliminaries

The main goal of our research described in this chapter is to improve automated GUI testing by providing means of reusing existing test cases across many application versions. The problem of GUI test-case maintenance has not gone unnoticed and several approaches have already been proposed. A first case study undertaken by Memon and Soffa (36) studies test case replayability for two versions of Adobe Acrobat Reader and an in-house clone of Microsoft WordPad. Further research (32) proposes inserting or removing events from unusable test cases in order to make them executable on the newer version, while in (21), Huang et al. detail an approach for increasing test suite coverage by replacing unfeasible test cases with similar ones obtained using a genetic algorithm.

Our research targets reusing GUI test cases by correctly identifying GUI changes and using this information to update existing test cases to work on newer versions of the target software. We accomplish this using a heuristic process able to match GUI windows and widgets across an application's versions. Correctly identifying these changes would allow perfect reconstruction for many unusable test cases, allowing more accurate regression testing of target applications. Preliminary work was presented by McMaster and Memon (25) who provide the following definition for the problem of matching equivalent widgets:

Definition 4.1 *Given two GUIs* G *and* G‘*, for each actionable GUI element* e_i *in* G*, find a corresponding element* e_j *in* G' *whose actions implement the same functionality. More formally, each widget from* G *and* G' *must be assigned to one of the following data structures (25):*

- Deleted - *Contains elements only found in the* older *GUI.*
- Created - *Contains elements only found in the* newer *GUI.*
- Maintained - *This is a set of GUI element mappings of the type* $(e_i \mapsto e_j)$ *that contains pairs of equivalent elements, with* $e_i \in$ G *and* $e_j \in$ G' *(From (25)).*

4.2 The Process

The goal of our matching process is to categorize all GUI elements belonging to the pair of GUI models provided into one of the three categories described in (25).. To

achieve this, our matching process works in three phases:

1. *Window Matching.* The first phase of the process is matching application windows. This is a required step to enable matching window elements themselves so the accuracy of the entire process is very sensitive to errors in window matching.

2. *Widget Matching.* All GUI widget matches are performed during this phase.

3. *Finalization.* This is the last step of the process, acting as a cleanup phase to make sure no GUI elements remain unclassified.

It is important to note that our process is able to match widgets only when they belong to matched application windows. This emphasizes the importance of correctly identifying matching windows and introduces limitations in identifying widgets moved across windows.

The matching process is customizable using an XML configuration file to provide the heuristic set to be used together with an execution strategy.

The heuristics are provided as a prioritized list $L_p = (h_1, h_2, ..., h_n)$, ordered so that $\forall 0 < i < j \leq n$, $P_i > P_j$, where P_i is used to denote the priority of the i_{th} heuristic.

The execution strategy controls the order in which the heuristic implementations are used and is responsible with running the provided heuristics in a way that achieves maximum accuracy. The current implementation provides the following strategies that can be used to control the first two phases of the matching process[1]:

- *Simple Execution Strategy.* Heuristics are executed in descending order of priority and each implementation is allowed to perform as many matches as it can. After the last heuristic runs the process moves to the next phase.

- *Priority Execution Strategy.* This strategy attempts to ensure that matches are always performed by the highest accuracy heuristic. To achieve this, the heuristics are executed in descending order of priority, always restarting the process when a new decision is taken. This strategy has the effect that matches found by lower priority heuristics can be used by high accuracy implementations in finding new matching elements.

[1]The third phase is only for cleanup and therefore not configurable

4.3 Implemented Heuristics

This Subchapter describes implemented heuristics that are ready to be deployed for widget matching. Implementations come in two flavours, according to the process phase they are used in:

- *Window Matchers.* These heuristics are used in the first phase of the process to match windows. Currently we only have one such heuristic available that is described in detail in the present section.
- *Widget Matchers.* These heuristics are used in the second phase of the process and are able to match GUI widgets. To achieve high accuracy, they must be provided with correctly matched window pairs. Our widget matching implementations are also described in this section.

WindowMatchingHeuristic[1]

This heuristic is used for matching application windows. Its accuracy is crucial as mistakes in window matching lead to none of the contained widgets being correctly paired. The implementation uses window titles to first match application root windows and then to examine remaining window pairs. If both GUI versions have only one window they are matched regardless of title.

PropertyValuesHeuristic

This is an object factory able to generate heuristic instances that perform matching by examining widget property values according to provided criteria. The current implementation supports three criteria types:

- *Equality.* Provided property values must be both non-null and equal.
- *Similarity.* Provided property values must be similar when diffed (52). An integer parameter is used to specify the maximum number of *add* or *delete* diff operations allowed with the values still considered similar. When employing this criteria, the total number of *add* and *delete* operations represents the match score. Eligible

[1] Heuristics are implemented using classes of the same name

widget pairs are then matched according to lowest match score. This approach allows for flexibility when property values change across GUI versions.

- *Nullity.* Both property values must be null.

Using these criteria a large number of heuristics can be generated, making it one of the workhorses of our process. The number or type of matching criteria is not limited. We can easily generate an implementation that matches widgets having equal *Class* property value, similar values for *Text* and *Icon* properties and null *Accelerator* value.

PropertyValuesHierarchyHeuristic

This is a heuristic factory that extends PropertyValuesHeuristic by creating implementations that search for widget matches only among children of already matched GUI elements. This implementation is due to our observations suggesting that many times matching GUI containers will contain matching children. The separate implementation also allows optimizing the matching code to work faster in the case of complex GUIs.

SingletonComponentHeuristic

This heuristic factory creates instances that can match widgets according to property value. What is different from implementations already discussed is the fact that matches are not arrived at by analyzing the value of the property in itself, but it's uniqueness. Instances are generated by providing a widget property name. The value associated with the provided property will then be used for matching. For example, an instance using the *Class* property would match a widget having *Class* value of *javax.swing.JButton* from an older window with another JButton instance only if they are both the only JButton's on their respective windows. The *Singleton* name hints to this behaviour, as each matched widget must somehow be a "singleton" on its window.

InverseHierarchyHeuristic

This implementation is our only non-configurable widget matching heuristic. It is useful for matching container widgets that were extensively modified across GUI versions. Given GUI elements $A \in GUI_{older}$ and $B \in GUI_{newer}$, this heuristic matches A with B if:

- All descendants of A have a matching GUI element that is a descendants of B.
- A has the same non-zero number of descendants as B.

FinalHeuristic

This heuristic was implemented to attend to the third phase of the match process. Its role is to assign all widgets that have remained uncategorized at the start of the last phase to the *Deleted* or *Created* structures, depending on the GUI model the unmatched widgets belong to. This final phase of the matching process cannot be customized and is implemented to ascertain that all GUI elements are classified.

4.4 Heuristic Metrics

In this Section we define metrics that are generally usable to assess the accuracy of a GUI element matching process. We start by defining a few metrics available using only oracle information:

Definition 4.2 Correct Decision Count (CDC). *The number of correct decisions for a given target application version pair. A decision is the action of classifying a GUI element as* Deleted *or* Created*, or an equivalent GUI element pair as* Maintained*.*

The CDC measure represents the total number of elements in the three defined data structures. Note that this differs from the number of widgets because a pair of equivalent elements in the *Maintained* structure is counted as one decision.

Definition 4.3 Correct Match Count (CMC). *The size of the* Maintained *structure. This represents the number of equivalent GUI element pairs.*

This measure indicates the degree of functional similarity between the studied GUIs. Most widgets having equivalents in the corresponding studied version is an indication that most application functionality is shared between the versions.

Definition 4.4 Dissimilar Widget Count (DWC). *The number of widgets changed between the studied versions. This includes all widgets in the* Deleted *and* Created *sets together with the number of widgets in equivalent pairs in which at least one property value that does not refer to size or location has changed.*

After running the process our automated evaluation harness analyzes the obtained results by comparing it to available oracle information and computes values for the metrics defined below:

Definition 4.5 Heuristic Correct Decision Count (HCDC). *The number of correct decisions taken by the process. Its value is a number between 0 (no correct decisions taken) and the CDC value available using oracle data.*

Definition 4.6 Heuristic Correct Match Count (HCMC). *The number of correctly determined equivalent GUI element pairs. This is the number of elements correctly assigned to the* Maintained *structure computed by the heuristic. Its value is a number between 0 (no correct matches) and the CMC value available using oracle data.*

Definition 4.7 Heuristic Correct Decision in Dissimilar Widgets Count (HCDDWC). *The number of correct decisions taken for dissimilar widgets. Its value is a number between 0 and the DWC available using oracle data.*

To better assess the accuracy of a heuristic run independently of GUI size we define the following measurements:

Definition 4.8 Heuristic Decision Rate (HDR). *The percentage of correct heuristic decisions, calculated as* $\frac{HCDC}{CDC}$.

Definition 4.9 Heuristic Match Rate (HMR). *The percentage of correct heuristic matches, calculated as* $\frac{HCMC}{CMC}$.

Definition 4.10 Heuristic Dissimilar Widgets Decision Rate (HDWDR). *The percentage of correct decisions for dissimilar widgets, calculated as* $\frac{HCDDWC}{DWC}$.

Our decision to define and implement three separate measurements, while providing no "standard" false positive/negative analysis is due to the peculiarities of our process which we believe has several distinct applications and its evaluation is better suited using particular measurements.

4.5 Case Study

This section introduces an extensive case study aiming to asses the accuracy and feasibility of our proposed heuristic process when applied to real-world GUI driven software.

We examine the result of running the process and devise a number of metrics to answer the following research questions:

1. What is the optimum heuristic process configuration?
2. How accurate is the process when applied to complex GUI software with the aim of enabling long term test-case maintenance and software visualization?
3. What type of matching errors can be expected and how can we limit their number?

4.5.1 A Highly Accurate Heuristic Set

We performed several experiments in order to answer research question (1). This required finding an optimal execution strategy together with the best performing set of heuristic implementations. However, due to both application GUI and heuristic implementation diversity it became clear that building a "best" heuristic set was not possible. Therefore we shift focus and use our data to build a highly accurate set of heuristics that provide a good balance between generality, accuracy and speed of execution.

The first step in building our high-accuracy set is using a combinatorial approach to build property-value based heuristic implementations using the prioritized properties in Table 4.1. To generate the implementations we run two loops over the property list. The outer loop controls the number of disregarded properties (n_{dp}) and takes values from 0 to the number of properties minus 2. The second loop controls which properties are disregarded traversing the table from bottom to top disregarding n_{dp} properties. At each step a new heuristic is generated and added to the priority list.

Accuracy	Property
Highly accurate	Icon Class Text Accelerator
Accurate	Index
Low accuracy	Width Height X Y

Table 4.1: Accuracy of recorded widget properties

To obtain the final version of our proposed heuristic set, we added *SingletonComponentHeuristic* and *InverseHierarchyHeuristic* implementations to the prioritized list of heuristics and removed implementations that consistently achieved poor decision rates.

Please note that all described artifacts are freely available for download on our project website (38).

4.5.2 Case Study Results

This section addresses research question (2) by presenting results obtained when applying the heuristic process described in the present Chapter to the 28 version pairs of FreeMind and jEdit in our software repository. Table 4.2 shows the aggregate results obtained by the heuristic set described in the previous section over the 28 application version pairs. Note that the presented results were obtained after composite widget subcomponents and demlimitation widgets were disregarded.

Measurement	FreeMind	jEdit	Total
Correct Decision Count	1799	8976	10775
Correct Match Count	1524	7461	8985
Dissimilar Widget Count	797	4115	4912
Heuristic Decision Count	1787	8953	10740
Heuristic Match Count	1505	7321	8826
Heuristic Correct Decision Count	1743	8436	10179
Heuristic Correct Match Count	1502	7194	8696
Heuristic Decision Rate	96.89%	93.98%	94.47%
Heuristic Match Rate	98.56%	96.42%	96.78%
Heuristic Dissimilar Widget Decision Rate	89.46%	80.46%	81.92%

Table 4.2: Heuristic process results

The most important results are found in the highlighted rows. Considering the time span of the targeted versions (7 years for FreeMind and 10 for jEdit) we consider decision rates well over 90% as very promising[1]. We believe match rates over 95% enable the implementation of long-lived GUI test cases adaptable across multiple application versions. Also, a high dissimilar widget decision rate shows that the process is able to match heavily modified widgets successfully, making it feasible for implementation in quickly evolving applications.

4.5.3 Heuristic Error Analysis

This Section details the encountered heuristic error types in our case study.

[1] We defined our metrics taking values between 0 and 1 for the sake of rigor, however we believe displaying percentages to be more intuitive

Detecting multiple changes in widgets

As expected, correctly matching widgets becomes more difficult when they are subjected to more changes across the studied versions. Figure 4.1 displays the *File* menu for two CVS snapshots of the FreeMind application. Due to multiple changes of the menu item that triggers printing our implementation did not correctly match the widgets, and they were classified as *Deleted* and *Created* during the final phase of the matching process.

Figure 4.1: Equivalent widget pair that was not correctly categorized as such

Detecting changes to complex widgets

We found that for certain complex components such as combo-boxes or tables we need to record more information to achieve consistency in providing correct decisions. Figure 4.2 provides such an example from a version pair of the FreeMind application. The problem stems from the combo-box used to zoom the displayed mind-map introduced in the later version. Lacking additional information, our process uses these components' indexes in their parent container for matching.

Changes beyond the GUI layer

FreeMind's evolution between January and February 2004 brought a change in one of the menu items within the *Edit* menu that could not be resolved solely by GUI analysis.

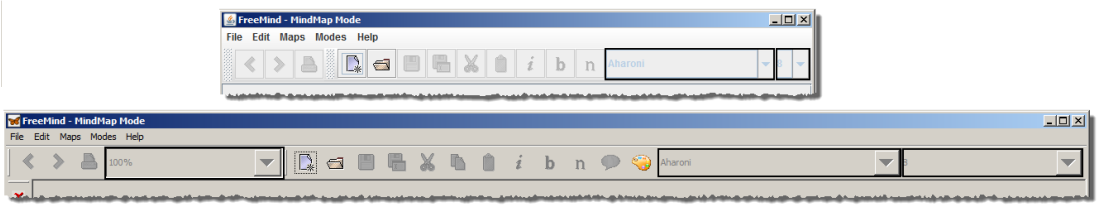

Figure 4.2: Mismatched combo-box widgets

This was due to changes at both GUI and event handling level which required source-code analysis to enable correct identification of equivalent widgets. Figure 4.3 shows the menu items with the heuristic decision highlighting the erroneously matched pair.

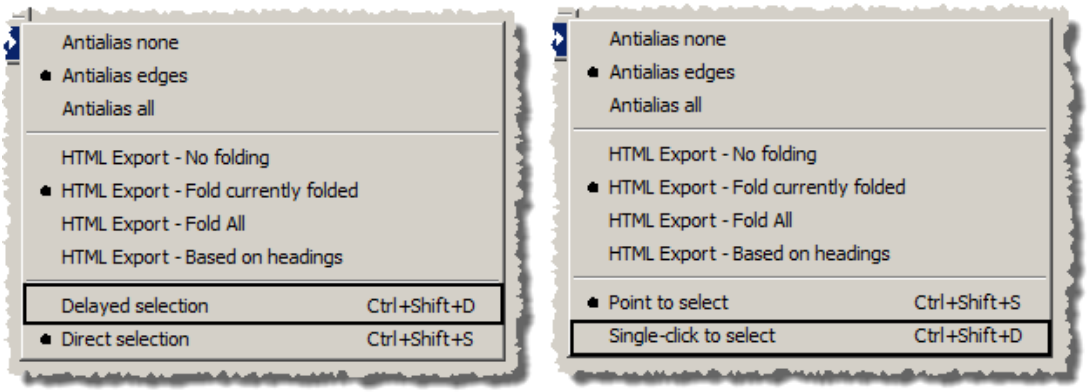

Figure 4.3: Not equivalent widgets having the same accelerator key

While rare, this type of error precludes from building a perfectly accurate process, as currently there are no code analysis techniques sufficiently advanced to be used in complex applications.

Icon file changes

One identified weakness of our process regards the handling the *Icon* property. Currently we use the icon file location for GUI element matching. This provides the advantage that icon file content can change without incurring penalties in matching performance as long as the icon file path is unchanged. However we found cases where both icon file path and contents were updated, breaking our matching implementations. Figure 4.4 shows an example using the "Print" button of two FreeMind versions in our repository.

Figure 4.4: Icon changes can lead to widgets not being adequately matched

Widget type changes

Figure 4.5 shows two groups of highlighted widgets from jEdit version 2.3final and 2.4final (left and right, respectively). These widgets were erroneously categorized in the *Created* and *Deleted* sets as they were not recognized due to extensive class changes.

Figure 4.5: Widget's type changing leads to mismatches

4.5.4 Threats to Validity

While we did our best to mitigate threats to the validity of our case study results we identified some aspects that need further detailing. First of all, due to the automated nature of our process it is possible that results obtained were skewed by software bugs. To mitigate this possibility we implemented multiple software safeguards that check every step of the process and hand-checked all oracle information. The second issue concerns generality. Both FreeMind and jEdit are complex applications that were employed in multiple previous studies (22, 56, 57, 58) related to GUI driven application testing. However, they represent a small fraction of all feasible GUI implementation. As such, there exists the possibility of these GUIs not being representative for all applications. To mitigate these aspects, our process is configurable and all artifacts are available for examination on our website (38).

4.6 Current Limitations

Although our matching process was designed for maximum flexibility, we identified some limitations that might preclude its use in some scenarios. Some of the identified issues stem from the tools we based our research on, while others are good candidates for future efforts. The most important limitations of our process are related to analyzing dynamic user interfaces, custom component implementations or pairs of GUIs that underwent major changes.

4.7 Conclusions and Further Work

A future direction of research consists of performing a more consistent case study by including several .NET and SWT based GUI applications. We would like to test our heuristic process using new application version pairs to have additional data about how the accuracy of the process changes when examining nightly or weekly builds of applications developed on different platforms. In addition, this would allow better understanding of how accuracy is affected by custom widgets implemented using other development libraries and platforms.

5

Management of GUI Testing

This Chapter presents our contributions to GUI application testing and GUI test case visualization. Subchapter 5.1 introduces the *GUI Test Suite Manager* tool that was built using components from our framework that we described in detail in Chapter 3. In Section 5.2 we undertake an extensive case study aiming to study GUI test case replayability in the context of our research in Chapter 4. Finally, we unite our seemingly diverging research paths by proposing an automated process of regression testing for GUI applications in Subchapter 5.3.

Our contributions in this Chapter are to be submitted for publication (9) and are listed within the introductory Chapter of this work.

5.1 A Software Tool for GUI Test Case Management

This section details our original contribution to GUI test case maintenance. We introduce new components in our software framework that was the subject of Chapter 3 and present a new software tool called *GUI Test Suite Manager* that integrates our original efforts presented in the previous chapters providing a novel approach to long term GUI test case management.

5.1.1 Measuring Test Coverage

Measurements such as statement and branch coverage have been in use for many decades and literature abounds with their advantages and shortcomings. However, GUIs are better defined by *events* rather than associated code, so a major direction of

research was studying how to devise new event-based coverage criteria. Memon et al. introduced important criteria for measuring test quality such as *event coverage*, *event interaction coverage* and its generalization *length-n event-sequence coverage*, described in (37). They also link event-based coverage criteria with code-based criteria using a case study that highlights how achieving sequence-n length coverage with $n = 2, 3$[1] leads to high code coverage.

In this Subchapter we propose a new approach that combines "traditional" code-based coverage measurements with information gained from static analysis obtained using the Soot framework and use Memon's definition of a GUI test case (30) presented in the course of the first Chapter. We first provide the following definitions that constitute the basis of our implementation.

Definition 5.1 Event static call subgraph. *Given a GUI event e_i, we define its static call subgraph as a subgraph of the application's statically computed callgraph that contains event e_i's application-level event handlers and all application methods reachable from them.*

Definition 5.2 Static step coverage. *Given a GUI test case T with event sequence $\{e_1, e_2, ..., e_n\}$, we define the* static step coverage *of event e_i, with $0 < i \leq n$ as the ratio of covered source code statements to all statement from the set of methods in e_i's event static call subgraph.*

The static step coverage is defined to provide a measurement of how well a test step exercises the set of *coverable* statements. With no static analysis tools we could only use code instrumentation to measure how many statements each test step actually covered during a test run but we were left with no information on which statements *could* have also been executed. Using applications such as Soot, we are able to provide such expectations apriori, without actually running the program.

Definition 5.3 Inclusive static step coverage. *Given a GUI test case T with event sequence $\{e_1, e_2, ..., e_n\}$, we define the* inclusive static step coverage *of event e_i, with $0 < i \leq n$ as the ratio of covered source code statements to all statement from the set of methods obtained by the reunion of all methods from the event static call subgraphs of events $E_{set} = \{e_1, e_2, ..., e_i\}$, with maximum achieved statement coverage taken into consideration for each method.*

[1] sequence-2 length coverage is actually event-interaction coverage.

Definition 5.4 Static test case coverage. *Given a GUI test case T with event sequence $\{e_1, e_2, ..., e_n\}$, we define the* static test case coverage *of test case T as the inclusive static step coverage of event e_n.*

5.1.2 The Test Suite Manager View

This Subchapter details the *Test Suite Manager View* component implemented with the aim of enabling management of GUI test cases created with the GUITAR framework. The main goal of our newly implemented component is to provide means to achieve point-and-click test case execution integrated with GUITAR and to provide feedback detailing achieved coverage at test suite, test case and test step level. Our component loads the GUITAR test suite together with coverage information for already executed test cases and processes it to obtain aggregate data about program runs. Figure 5.1 shows the *Test Suite Manager* component within the *GUI Test Suite Manager* software tool, described in the following Section.

5.1.3 The Code Coverage View

This view is an extension of our previously introduced *Call Graph View*, detailed in Subchapter 3.3.3. Our rationale when implementing this component is to provide a detailed view of static code coverage for each GUI test case step.

Figure 5.1 shows the Code Coverage View in action within our *GUI Test Suite Manager* tool. Note that like its sibling Call Graph View, the Code Coverage View component can show a GUI event's static call subgraph and is therefore useful to examine static coverage information for test steps. The color coding is consistent with the *Test Suite Manager* component, with the amount of green representing the ratio of executed statements over their total number for the given method. Again, we use different shades of green to represent code that was executed during the step proper or during preceding test steps.

5.1.4 The GUI Test Suite Manager Tool

This tool was implemented to provide means of managing the GUI testing process execution and to provide a readily accessible assessment of key measurements that characterize employed testing suites.

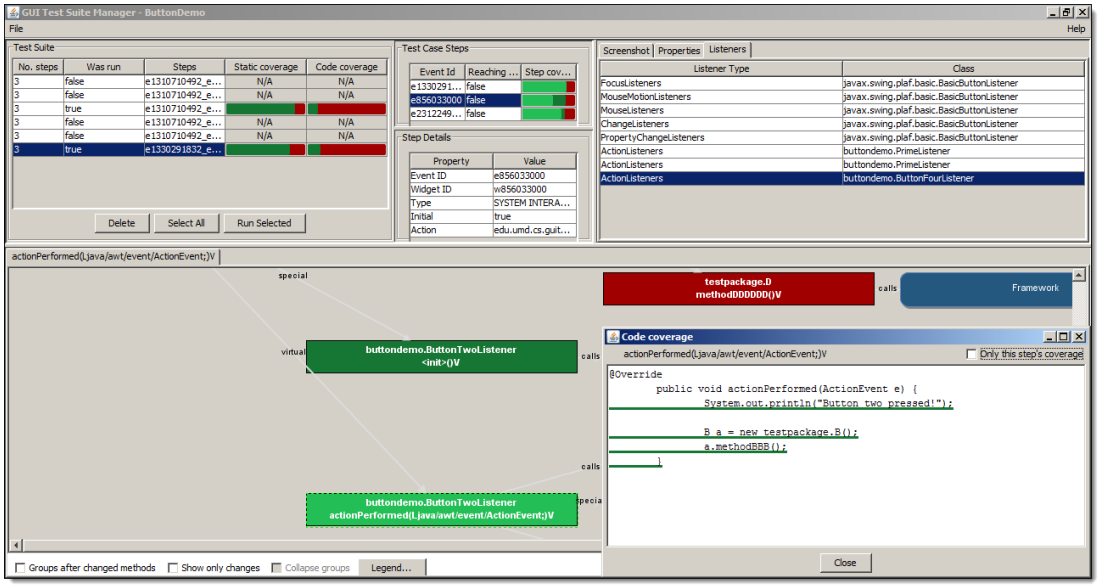

Figure 5.1: GUI Test Suite Manager Tool

The tool's user interface is seen in the screen capture in Figure 5.1, where the reader can recognize the three constituting components. Similar with jSET, the GUI Test Manager uses our Project system described in the 2nd Chapter as input data. The Widget Information View is used to provide users with valuable knowledge regarding GUI elements. Each time the user selects a GUI test step using the Test Manager View, its properties and screenshot are displayed to enable further examination. Also, the newly implemented Code Coverage View shows associated coverage for already executed test steps, allowing a detailed examination of the target event's step and inclusive step static coverages.

5.2 A Study in GUI Test Case Replayability

In this Subchapter we present a case study that aims to assess GUI test case replayability for complex real-world applications. In the present study we reuse the GUI driven applications from our repository introduced in the 2nd Chapter. We synthesize our research as an attempt to provide answers for the following research questions: (1) How do changes in the application GUI affect test case replayability? (2) How effective our heuristic process is at maintaining test cases replayable?

This Subchapter is divided in two sections. The first section presents a "best case" concerning GUI test case replayability where we use oracle information built for our GUI element matching process to assess how many test cases survive GUI changes in a cross-sectional study. In the second section we use our highly accurate heuristic set built in Subchapter 4.5.1 to determine how well our process is suited to repairing existing GUI test cases so they work on newer versions of the target application, again using a cross-sectional approach.

We use GUITAR's Test Case Generator to obtain test cases for the 28 FreeMind and jEdit versions in our repository. We then simulate their execution using the available GUI model and event-flow graph to assess their replayability. As Previous studies have established strong links between GUI test case length and fault-detection capability (Xie and Memon (55)), we decided to generate all length-2 test cases together with 5000 length-3 and 5000 length-4 randomly generated test cases, totalling a number of 451062 test cases for all application versions.

5.2.1 Having Perfect Information

This section provides an answer for research question (1). It examines an ideal situation where perfect knowledge on equivalent GUI elements is available. In our case, this is represented by available oracle information that was built as part of the case study in Chapter 4. Our aim is to study the replayability of existing GUI test cases as target applications evolve. For this, we divide our generated test cases into four categories:

1. *Replayable using widget Id.* This simulates how test cases can be replayed by less sophisticated tools that identify widgets using identifiers.

2. *Replaying using Widget Matching.* This category represents existing test cases that can be exactly replayed on the new version of the application. This requires actioned elements in both test and reaching steps to have equivalents on the newer version of the GUI and their sequence to be valid according to the newer version's event-flow graph.

3. *Repairable using Widget Matching.* Here we relax the event flow graph condition imposed in the previous step and only require the existence of equivalent widgets on the new application version.

4. *Unrepairable.* This last category comprises those test cases that cannot be repaired.

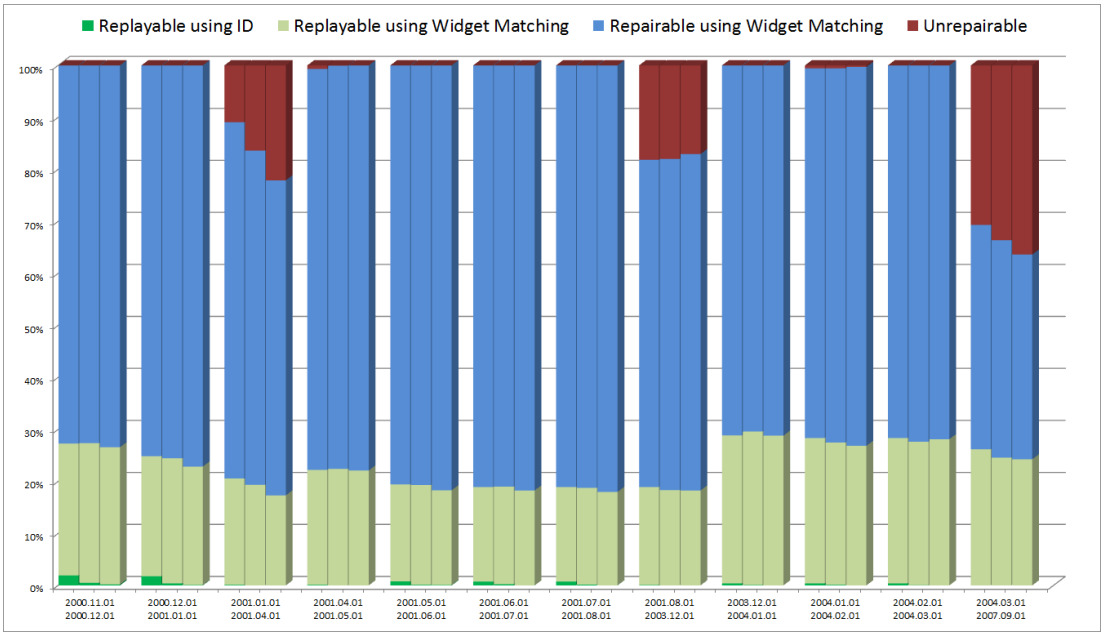

Figure 5.2: FreeMind test cases, ideal replayability

Figure 5.2 summarizes our results for the FreeMind application. Each pair of application versions is represented with 3 columns. From left to right, they display information about replayability of test cases having lengths of 2,3 and respectively 4. Note that having perfect information available leads to most test cases being at least "repairable" according to the classification above.

Figure 5.3 shows the results for the jEdit text editor. The main difference is that many more test cases regularly become unrepairable between application versions. While the time elapsed between studied jEdit versions is comparable with FreeMind, jEdit is a more complex application, having between 12 to 16 recorded windows.

5.2.2 Using Our Heuristic Process

In this section we repeat our experiment presented above, but this time we employ our imperfect heuristic process to assess test case replayability to provide an answer to research question (2).

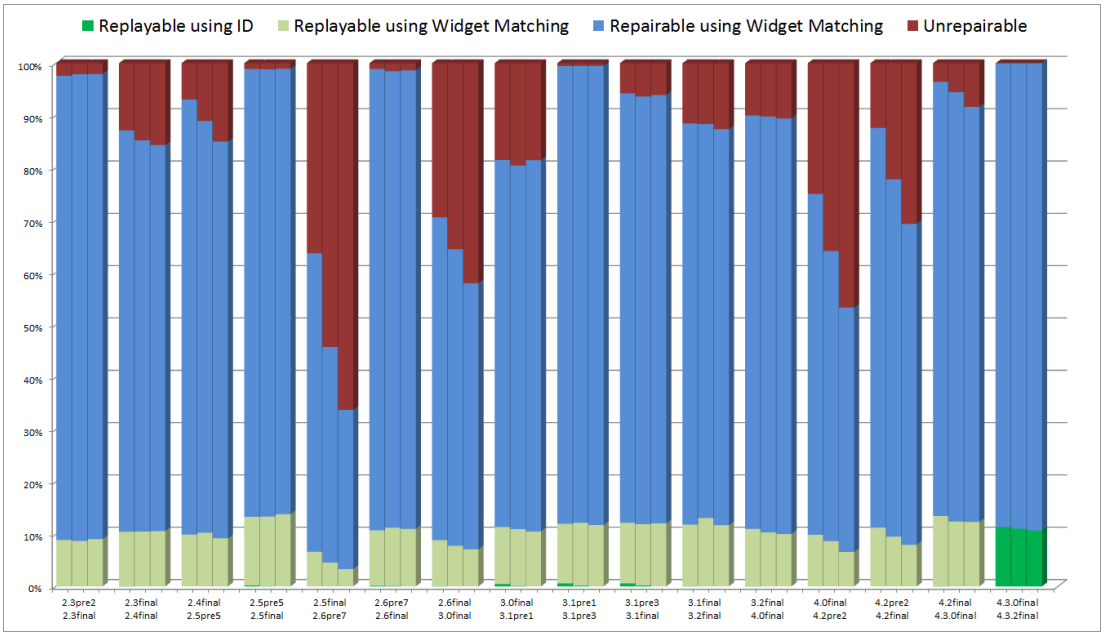

Figure 5.3: jEdit test cases, ideal replayability

Figure 5.4 shows the results of our cross-sectional study on test case replayability for the FreeMind application. As expected, our results closely match the ideal situation presented in the previous section. Since our process achieved 98.56% accuracy in matching equivalent widgets, it was already close to a perfect heuristic, with most test cases being in the same category as was the case with perfect information.

Figure 5.5 shows test case categorization information for the jEdit text editor. As our matching rate was lower with jEdit (at 96.42%), we expected more degradation of test case replayability when compared with oracle information. Unfortunately, in four versions most test case were rendered mostly unrepairable.

Also, we must note the effect test case length has on replayability, as longer test cases are significantly more prone to breaking across application versions. However, taking into account the 10 year timespan between the first and last studied version leads us to believe that our results show that building long-lived GUI test cases is possible, but special care must be taken when significantly changing the user interface.

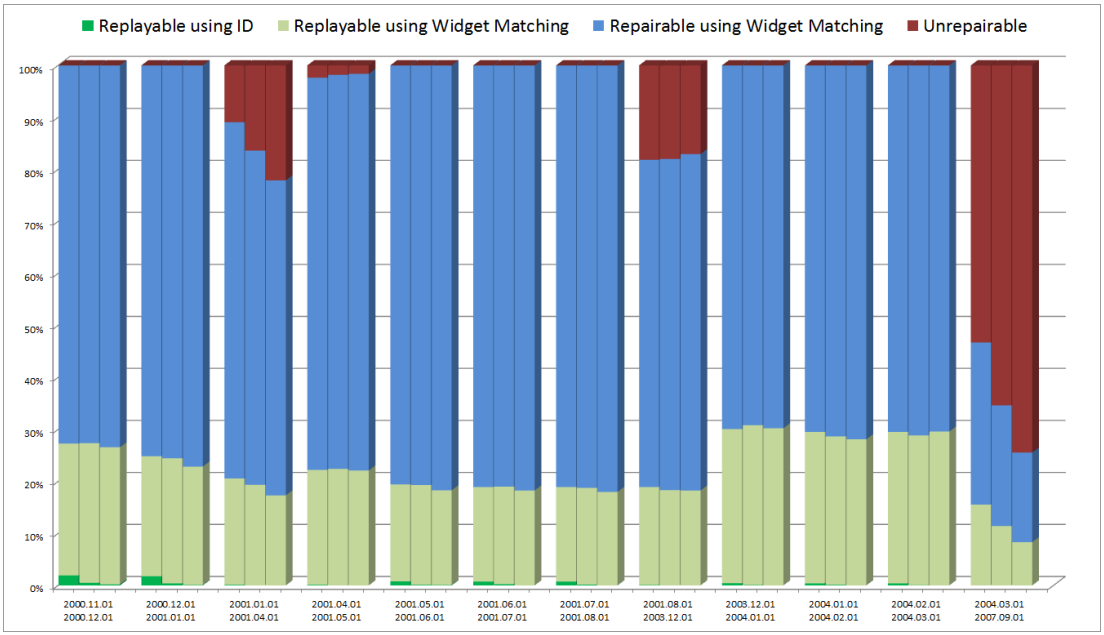

Figure 5.4: FreeMind test case replayability using heuristics

5.2.3 Threats to Validity

A first threat concerns our chosen target applications. Although they were employed in multiple studies targeting software testing, we cannot generalize our obtained results to all GUI-driven applications. Other applications might have peculiarities that would drastically affect obtained results.

A second threat concerns the process used to obtain our data. Due to the large number of test cases, we have not actually replayed them on the newer versions of the target software but performed a static analysis on their replayability by taking into account GUI and event flow graph data. While GUITAR's Test Case Replayer uses the same data to actually replay test cases, there might exist certain errors in the recorded GUI model, event flow graph or other application specific peculiarities that might preclude the replay of some of the generated test cases.

5.2.4 Current Limitations

Aspects that can be improved by future efforts regard the implementation of more event-centric coverage information(55), better assessment of test case results and providing a semi-automated approach for building test cases.

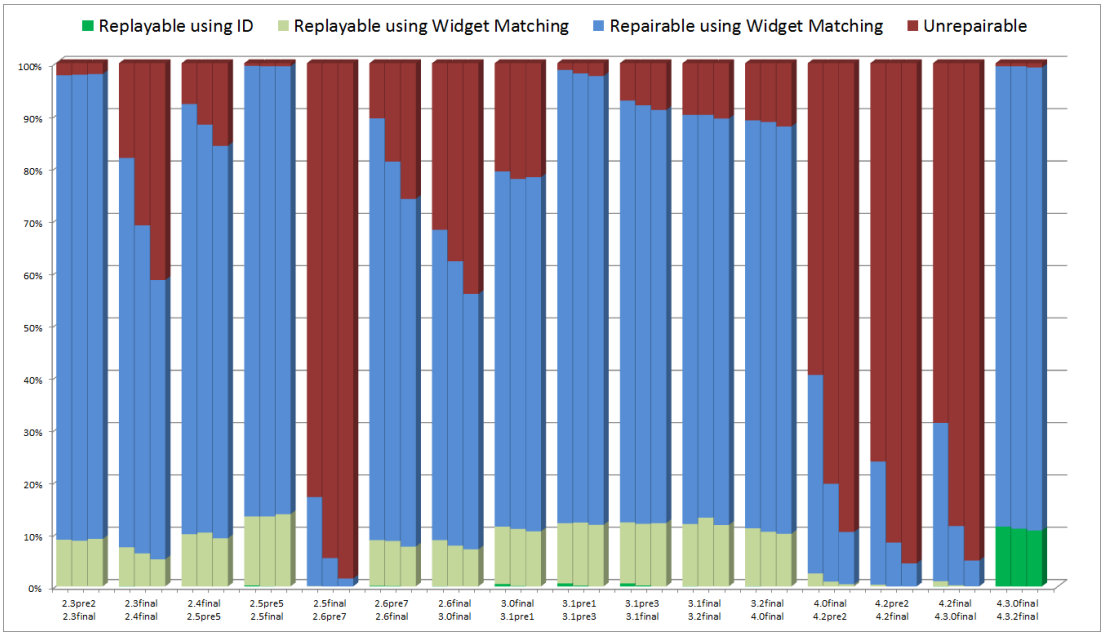

Figure 5.5: jEdit test case replayability using heuristics

5.3 Integration in a Production Environment

In this Subchapter we present an approach to integrate our research efforts in the development of a GUI-driven software application with the aim of implementing a trackable regression testing system. Figure 5.6 shows the steps of our proposed process from left to right.

The proposed system works in the five presented steps. The steps are detailed as follows:

1. *Nightly build.* This triggers the regression testing process. Note that we use nightly as a convenience term, as the proposed process can work at any given timespan between subsequent application builds.

2. *Create project.* Chapter 2 introduced the Project system used to build our software repository together with implemented means of automation that enable building Project instances for all application builds.

3. *Repair test cases.* This step consists of running our implementation of the heuristic process using an accurate configuration on the existing test suite to adapt them to the new user interface.

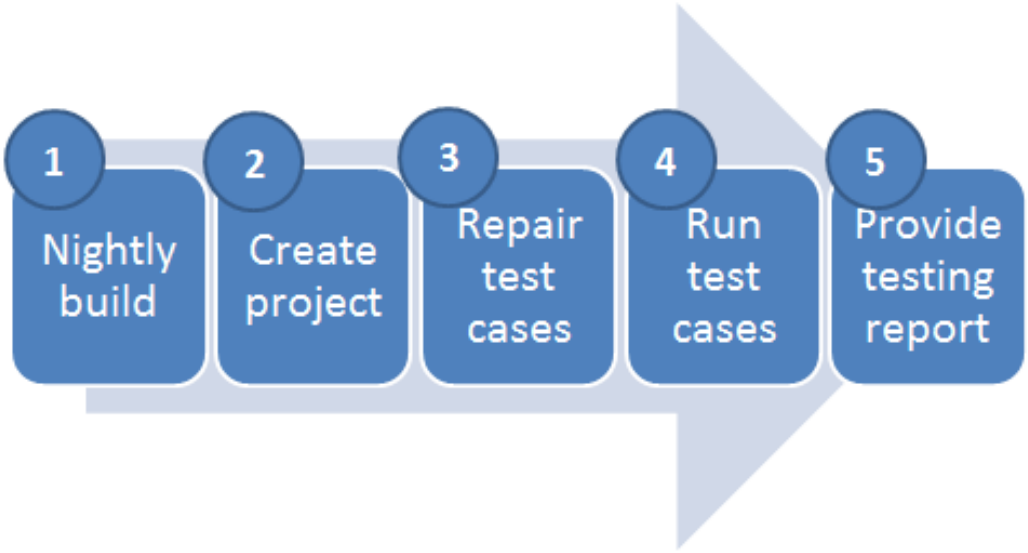

Figure 5.6: Regression Testing Process

4. *Run test cases.* This step uses GUITAR's Test Replayer component to run the repaired test cases on the new build and record the state of the target GUI.

5. *Provide testing report.* Our toolset can provide valuable information regarding test step and test case coverage. This makes it suitable for finding application areas that were not thoroughly tested and to guide the creation of test cases to achieve best coverage.

Our process is not a first of its kind. We find a similar effort described in Memon's PhD thesis (30) where a regression testing process is proposed. The approach is then refined in (33), where the DART process is introduced. Comparing these initial approaches to our process we find they lack a project system with associated tools that enable visualization and analysis of the target application. Also, they lack a way to repair existing test cases, relying on GUI elements to have constant names by which they are recognized instead.

A more recent approach is found in Xie's PhD thesis (53), where she proposes a continuous process for GUI testing, which has an important regression component based on the GUITAR framework. We believe approaches proposed in (53) are integrateable with our presented process and can be harnessed in software development to enable

automated GUI smoke testing, regression testing and stand-alone version testing using the crash oracle presented by Xie (53).

5.4 Conclusion and Future Work

This Chapter provides a holistic view on our research efforts on GUI application visualization and testing. We presented a new instantiation of our component framework, the *GUI Test Case Manager* tool and we detailed provided visualizations. We followed through with our work in Chapter 4 and studied how effective our proposed process is for repairing GUI test cases to work on new application versions.

Concerning future efforts, our desire is to study how our developed tools can be integrated in a continuous testing environment such as the one presented by Porter et al. (40). Also, we believe effort should be dispensed toward building new tools to allow user-guided test case generation. Such tools will enable bridging the man-machine gap by allowing the creation of effective test suites that can be executed and evaluated automatically, bringing us closer to Bertolino's "dream" of 100% automated testing (4).

6

Conclusions

Our research targeted two areas. The first is GUI-driven software visualization with an emphasis on enabling new ways of analyzing and visualizing such applications. Chapter 3 introduced our component framework which consists of multiple components supporting the development of such tools. The first tool we described was jSET, which proposed a holistic approach to program visualization by incorporating analysis tools at each application level, namely user interface, underlying call relations and program source code. Another implementation using our framework was detailed in Chapter 5: the GUI Test Suite Manager application that enables GUI test case management together with providing advanced visualization capabilities for detailing test case execution.

The second pursued avenue of research targeted regression testing of GUI applications. In Chapter 4 we introduced a new heuristic process for GUI element matching, a brainchild of McMaster and Memon's (25) paper. We also perform an extensive case study where we employed newly defined metrics to assess the accuracy of our process. More importantly, we perform a thorough analysis of error classes encountered and we proposed solution for most such situations.

Our research directions unite in Chapter 5, where we study the efficiency of our heuristic matching process on repairing actual GUI test cases for complex open source software in our repository. We compare our best results with the "best case" scenario obtained using oracle information and we show that our process is effective as the basis of a GUI regression testing process.

Of course, the presented research would lose all meaning without suitable target applications. For this reason, we used the 2nd Chapter to present our repository of

complex open-source applications that we used throughout our studies.

It is our belief that the work presented here opens interesting new avenues in both directions researched. First of all, we found that software visualization benefits greatly from advanced analysis tools such as Soot and GUITAR which currently are mainly used in academic research. A future direction is to integrate our component framework into well-known IDE's such as Eclipse as plugins. This would enable their deployment without altering existing industrial processes and would lead to acceptance and use among developers. Another aspect regards our work on GUI test case maintenance. Our plans are to integrate our heuristic process with the open-source GUITAR framework to enable its automatic application whenever changes are detected within the target application. More so, we aim to develop a user-friendly test environment to enable long-term GUI test case management which includes capabilities for semi-automated generation of GUI test cases using AI techniques (30, 35). We believe that integrating such tools with popular IDE's such as Eclipse will speed the adoption of newly developed testing methodologies and processes in the industry, ultimately leading to software that is better and cheaper.

Bibliography

[1] Bach, J. Test automation snake oil. *Windows Tech Journal* (1996), 40–44.

[2] Baresi, L., and Young, M. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[3] Bertolini, C., and Mota, A. A framework for gui testing based on use case design. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2010), ICSTW '10, IEEE Computer Society, pp. 252–259.

[4] Bertolino, A. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering* (Washington, DC, USA, 2007), FOSE '07, IEEE Computer Society, pp. 85–103.

[5] **Arthur-Jozsef, M.** A heuristic process for GUI widget matching across application versions - abstract. In *Abstracts of MaCS 2012*.

[6] **Arthur-Jozsef, M.** jSET - Java Software Evolution Tracker. In *KEPT-2011 Selected Papers*, Presa Universitara Clujeana, ISSN 2067-1180.

[7] **Arthur-Jozsef, M.** jSET - Java Software Evolution Tracker - extended abstract. *KEPT 2011 Conference, Cluj Napoca* (July 2011).

[8] **Arthur-Jozsef, M.** A heuristic process for GUI widget matching across application versions. *Annales Universitatis. Scientiarum Budapestinensis, Sectio Computatorica* (2012).

[9] **Arthur-Jozsef, M.** An initial study on GUI test case replayability. *IEEE International Conference on Automation, Quality and Testing, Robotics AQTR 2012, Cluj-Napoca - submission pending* (2012).

[10] **Arthur-Jozsef, M.** A software repository and toolset for empirical software research. *Studia Informatica UBB, Cluj-Napoca - submitted* (2012).

[11] Brooks, P., Robinson, B., and Memon, A. M. An initial characterization of industrial graphical user interface systems. In *ICST 2009: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2009), IEEE Computer Society.

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

[13] Cabral, G., and Sampaio, A. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci. 195* (January 2008), 171–188.

[14] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (London, UK, UK, 1995), ECOOP '95, Springer-Verlag, pp. 77–101.

[15] Goodenough, J. B., and Gerhart, S. L. Toward a theory of test data selection. *SIGPLAN Not. 10* (April 1975), 493–510.

[16] Hackner, D., and Memon, A. M. Test case generator for GUITAR. In *ICSE '08: Research Demonstration Track: International Conference on Software Engineering* (Washington, DC, USA, 2008), IEEE Computer Society.

[17] Hammontree, M. L., Hendrickson, J. J., and Hensley, B. W. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1992), CHI '92, ACM, pp. 431–432.

[18] Hass, A. M. J. *Guide to Advanced Software Testing.* Artech House, Inc., Norwood, MA, USA, 2008.

[19] Hou, D. Studying the evolution of the eclipse java editor. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange* (New York, NY, USA, 2007), eclipse '07, ACM, pp. 65–69.

[20] Hou, D., and Wang, Y. 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* (New York, NY, USA, 2009), CASCON '09, ACM, pp. 122–135.

[21] Huang, S., Cohen, M. B., and Memon, A. M. Repairing gui test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation* (Washington, DC, USA, 2010), ICST '10, IEEE Computer Society, pp. 245–254.

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

[23] Lhotak, O. Spark: A flexible point-to analysis framework for java. Tech. rep., McGill University, Montreal, 2002.

[24] Lhotak, O. *Program analysis using binary decision diagrams.* PhD thesis, Montreal, Que., Canada, Canada, 2006. AAINR25195.

[25] McMaster, S., and Memon, A. M. An extensible heuristic-based framework for gui test case maintenance. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 251–254.

[26] Memon, A. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *In Proceedings of The 10th Working Conference on Reverse Engineering* (2003), pp. 260–269.

[27] Memon, A., and et al. What test oracle should i use for effective gui testing? In *PROC. IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'03* (2003), IEEE Computer Society Press, pp. 164–173.

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

[29] Memon, A., and Xie, Q. Using transient/persistent errors to develop automated test oracles for event-driven software. In *Proceedings of the 19th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 186–195.

[30] Memon, A. M. *A comprehensive framework for testing graphical user interfaces.* PhD thesis, 2001. AAI3026063.

[31] Memon, A. M. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability 17*, 3 (2007), 137–157.

[32] Memon, A. M. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol. 18* (November 2008), 4:1–4:36.

[33] Memon, A. M., Banerjee, I., and Nagarajan, A. DART: A framework for regression testing nightly/daily builds of GUI applications. In *Proceedings of the International Conference on Software Maintenance 2003* (Sept. 2003).

[34] Memon, A. M., Pollack, M. E., and Soffa, M. L. Automated test oracles for guis. *SIGSOFT Softw. Eng. Notes 25* (November 2000), 30–39.

[35] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng. 27*, 2 (2001), 144–155.

[36] Memon, A. M., and Soffa, M. L. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ACM Press, pp. 118–127.

[37] Memon, A. M., Soffa, M. L., and Pollack, M. E. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2001), ACM Press, pp. 256–267.

[38] Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv. 33* (March 2001), 31–88.

[39] Nguyen, D. H., Strooper, P., and Suess, J. G. Model-based testing of multiple gui variants using the gui test generator. In *Proceedings of the 5th Workshop on Automation of Software Test* (New York, NY, USA, 2010), AST '10, ACM, pp. 24–30.

[40] Porter, A. A., Yilmaz, C., Memon, A. M., Schmidt, D. C., and Natarajan, B. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Trans. Software Eng. 33*, 8 (2007), 510–525.

[41] Ramler, R., and Wolfmaier, K. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test* (New York, NY, USA, 2006), AST '06, ACM, pp. 85–91.

[42] Robinson, B., and Brooks, P. An initial study of customer-reported gui defects. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 267–274.

[43] Staats, M., Whalen, M. W., and Heimdahl, M. P. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 391–400.

[44] Strecker, J., and Memon, A. M. 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, 2008), IEEE Computer Society.

[45] Sundaresan, V. Practical techniques for virtual call resolution in java. Tech. rep., McGill University, 1999.

[46] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), CASCON '99, IBM Press, pp. 13–.

[47] Website. http://guitar.sourceforge.net/. Home of the GUITAR toolset.

[48] Website. http://www.sable.mcgill.ca/soot/. Soot home at McGill University.

[49] Website. https://svn.sable.mcgill.ca/wiki/index.cgi/SootUsers. Soot's list of users.

[50] Website. http://sourceforge.net/projects/freemind/. Home of the FreeMind project.

[51] Website. http://sourceforge.net/projects/jedit/. Home of the jEdit project.

[52] Website. http://code.google.com/p/google-diff-match-patch (Home of an implementation for diff-match-patch).

[53] Xie, Q. *Developing cost-effective model-based techniques for gui testing.* PhD thesis, College Park, MD, USA, 2006. AAI3241432.

[54] Xie, Q., and Memon, A. M. Model-based testing of community-driven open-source gui applications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 145–154.

[55] Xie, Q., and Memon, A. M. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol. 16* (February 2007).

[56] Yuan, X., Cohen, M. B., and Memon, A. M. Gui interaction testing: Incorporating event context, 2011.

[57] Yuan, X., and Memon, A. M. Alternating gui test generation and execution. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 23–32.

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