Definition and Evaluation of Mutation Operators for GUI-level Mutation Analysis

Rafael A.P. Oliveira
ICMC/USP
University of Sao Paulo
Sao Carlos, Brazil
rpaes@icmc.usp.br

Emil Alégroth
Software Eng. and Tech.
Chalmers University
Gothenburg, Sweden
emil.alegroth@chalmers.se

Zebao Gao
Dept. of Computer Science
University of Maryland,
College Park MD, USA
gaozebao@cs.umd.edu

Atif Memon
Dept. of Computer Science
University of Maryland,
College Park MD, USA
atif@cs.umd.edu

***Abstract*—Automated testing has become essential in software industry to meet market demands for faster delivery and higher quality software. Testing is performed on many levels of system abstraction, from tests on source code to *Graphical User Interface* (GUI) tests. New testing techniques and frameworks are also continuously released to the market.**

Mutation analysis has been proposed as a way of assessing the quality of these new test techniques/frameworks as well as existing test suites in practice. The analysis is performed by seeding defects, referred to as mutants, into the system under test with the assumption that a technique/test suite of high quality will "kill" the mutants. However, whilst support for mutation analysis exists for test techniques that operate on on lower levels of system abstraction, i.e. method-level mutation operators, the support for GUI-level mutation analysis is currently lacking.

In this paper we perform an empirical analysis of 18 GUI-level mutation operators defined in our previous work and compare their efficiency and comprehensiveness to state-of-practice lower level mutation operators. The main findings of our analysis are (1) that traditional method-level mutation operators are not precise enough for GUI-level mutation; (2) the defined GUI-based mutation operators provide comprehensive support for GUI-level mutation; and (3) GUI-based mutation operators can be automated but are challenged by the dependencies between GUI widgets.

***Keywords*-mutation testing, GUI testing, Graphical User Interface, mutation operators, software testing[1]**

I. Introduction

GUI(*Graphical User Interface*)-based applications are systems in which the interaction between underlying code and users occurs through a pictorial human-machine interface [1]. GUIs alleviate the users' efforts on exploring resources and functionality of software systems, increasing their productivity and saving time. Users can interact with the underling code of a GUI-based application through "events": mouse clicks, mouse drags, keyboards shortcuts or commands, object manipulation, etc [2]. Currently, GUI-based applications are ubiquitous in practice and well designed GUIs are often key for the success of software application.

Due to the their importance, today's GUI-based software has attracted the attention of both practitioners and researchers. Software developers are spending a significant amount of time modeling, implementing and testing software systems on a GUI-level of abstraction. Additionally, significant research efforts are spent on developing and analyzing novel automated test techniques and strategies [3]. Research warranted by automated and systematic GUI-level testing transitioning from a want to an essential need for companies to keep up with the software market's demands for faster software delivery and higher software quality [4].

New testing techniques are continuously developed on different levels of abstraction. Most of these techniques are addressed for specific purposes. In industrial scenarios, companies develop their own GUI supporting tools, limiting the share of knowledge in this research area. Generic solutions for GUI level testing are lacking [3]. Regardless, there is a need for assessing the quality of these new techniques, which has been proposed be achieved with mutation testing concepts [5].

Mutation testing [5] is a promising strategy to support the evaluation of novel GUI testing techniques and strategies for generating testing data for GUI-based applications. Mutation is about comparing slightly modified versions of a program (mutants) and its original code, observing the program's behavior against these modifications [6]. *Mutation Operators* (MO) are instruments to include modifications to specific statements of the source code, generating the mutants. Further, mutation is a fault-based testing technique in which mutation operators are syntactic or paradigm-based simple rules used to change the original program, creating mutated, faulty, SUT versions. Using a supporting tool, through a test set, testers then compare the SUT's (*Software Under Test*) outputs and the mutants' outputs, "killing" the mutants with different outputs. Mutation analysis is useful not only to assess the fault-finding effectiveness of test sets, but also to evaluate the application of new testing approaches for specific domains. As such, applying novel approaches on mutant programs represents a way to evaluate the techniques' benefits and drawbacks.

The body of knowledge on mutation analysis includes an adequate support for the technique's applicability for mutation of the underlying system code and its functionality. However, regarding GUI testing, to guarantee the manifestation of mutants on an SUT's pictorial GUI, we stipulate that the concept of mutation analysis needs to be adopted. To do so, special MOs, derived from the general mutation

[1]Rafael is supported by FAPESP 2012/06474-1

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th International Workshop on Mutation Analysis (Mutation 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

analysis paradigm, are required. GUI-testing, in contrast to testing of low-level underline code, operates across widgets, properties (and values). GUI testing is executed with scenarios of events that represent particularities that must be considered to evaluate the SUT's requirement conformance [7]. Sequence of events, properties, and values of properties compose test data for GUI-based applications, requiring the application of specific strategies [3]. Traditional mutation tools do not provide sufficient support to measure the effectiveness of novel testing techniques for the GUI level of abstraction. In previous work [8], we identified initial support for this claim when the generally proposed MOs and associated tools proved insufficient to create suitable mutants to evaluate the efficiency of the GUI-based testing techniques component-based and *Visual GUI Testing* (VGT) [4]. Thereby warranting a need to adapt existing approaches of mutation analysis for GUI testing.

In this paper we build on our previous work and present a multi-step empirical study where we evaluate the effectiveness of a set of 18 GUI-based MOs and compare them against traditional method-level MOs[2]. Results of the comparison show that our operators are comprehensive to create GUI level mutants and that method-level MOs are difficult to guide to create suitable GUI-level mutants. Further, the study is performed on both a smaller calculator application as well as a larger *Open-Source Software* (OSS).

The contributions associated with this paper are:

- An empirical analysis on the fault-seeding effectiveness of traditional MOs to explicitly generate GUI-level mutants in comparison to the specific GUI-level MOs;
- A framework for automating the generation of mutated GUIs from the MOs proposed in [8];
- Proof of concept of the applicability of the automated MOs on the GUI-level of a real complex OSS application, eliciting their benefits and weakness.

The continuation of this paper is organized as follows: Section II presents basic concepts on GUI-based applications, and GUI testing; Section III details the MOs for GUI-level mutation analysis; three research questions, a pilot study, and a proof-of-concept are described in Section IV; in Section V we present further discussions about the findings outlines from the proof-of-concept and the potential answers to our research questions; Section VI discuss threats associated to our study; finally, Section VII presents the final remarks and the conclusions from this study.

II. Background

This section presents fundamental concepts associated with this paper: GUI-based applications, and some concepts on testing them; and MOs for GUI-level mutation analysis.

A. GUI-based applications

GUI-based applications have some different particularities in comparison to regular applications. GUI accepts as input events from users or from the system by itself. Then, from these events, they generally, produce deterministic graphical outputs. The representation of a GUI instance is defined by a state, which is modeled as a triple (*Ws*, textitP, textitV). In this scenario, let $W_s = W_{s1}, W_{s2}, ..., W_{sn}$ be a set of widgets. Each widget has a set of properties $P_s = P_{s1}, P_{s2}, ...P_{sn}$ and, finally, each property can assume a value in its contemplation $V_s = V_{s1}, V_{s2}, ...V_{sn}$ [1].

B. GUI testing

GUIs' particularities lead to some test related challenges. A GUI testing activity must have test cases modeled as sequences of user inputs combined in events [1]. Testers design a sequence of inputs to exercise GUI's widgets (e.g., click on buttons, fill text fields out, resizes, changing orientation, etc), revealing potential faults [3]. Faults, in the GUI context, can be associated to the underline code as well as the visual appearance of the interface. In this context, several challenges arise from the GUI testing philosophy: generation of events, test oracle design, and defining what to test on the GUI perspective [3]. Depending on the SUT, the tester has to define a set of adequate criteria to define whether the SUT behaves according to its specification.

Regarding the automation support, [4] have stated that automated GUI testing strategies and tools can be classified into three different chronological generations: (1) the first generation consists of approaches based on screen coordinate-based interaction with the SUT such as Record/Playback; (2) the second generation of tools relies on software component/widget-based SUT interaction; and (3) *Visual GUI Testing* (VGT) that combines image recognition and script-based high-level testing.

The first generation tool's have been abandoned due to poor robustness and high maintainability. Second generation tools are common in practice, support high-level testing, have quick execution times, etc. However, these tools operate against the GUI model, i.e. the underlying code that defines the visual appearance of the GUI, rather than the actual GUI. Therefore, these tools can miss GUI level defects if they are not also represented in the model. Visual GUI testing (third generation) does not have this problem since the technique operates against the pictorial GUI as shown to the user. Hence, emulating user's behavior and exploring computational vision strategies [4].

Due to the competitiveness in the software market, GUI-based applications need for more sophisticated testing strategies and support. According to [3], several automated GUI testing techniques have been proposed and evaluated in the last decade. As a consequence, researchers and practitioners need for methods to support assessing the quality of their automated test cases and new test techniques. Section III presents a set of MOs recommended for GUI-level mutation analysis.

III. Mutation operators for GUI-level mutation analysis

As opposed to their effect on regular source code, traditional MOs, which mimic typical programming errors, have few

[2]In the scope of this study, traditional method-level mutation operators are those able to modify operands and statements creating slight versions of the original code.

influence on the GUI level, stifling their ability to be used for GUI-level mutation analysis. This section presents in details a set of MOs particularly designed to mimic GUI-level faults.

A. MOs for GUI-level mutation analysis

Initially proposed in previous work [8], Table I presents a set of platform-independent mutation operators that can be implemented in different GUI libraries. The mutation operators considered in the scope of this study can be divided into three different classes: (1) "removing"; (2) "adding"; and (3) "modifying" code in the SUT. Often GUI faults result in missing components, wrong position of components, blurred or distorted images, unexpected screens, etc. Then, [8] match different classes of MOs to a potential fault of the SUT that is propagated through the GUI.

TABLE I: MOs for GUI testing techniques

#	Class	Mutant Operator	Acronym
1	Rem.	–Remove Existing Widget	REW
2		–Set Widget Invisible	SWI
3		–Remove Existing Listener	REL
4	Adding	–Add Identical Widget	AIW
5		–Add Similar Widget	ASW
6		–Add Different Widget	ADW
7		–Add Another Listener	AAL
8	Modifying	–Expand/Reduce size of Windows and Widgets will Auto-adjust Their Sizes	EWWAR/ RWWAR
9		–Expand size of windows and widgets will not auto-adjust their sizes	EWWNAR/ RWWNAR
10		–Reduce size of windows to hide widgets	RWHW
11		–Modify location of a widget to a proper location	MLWP
12		Modify location of a widget to edges of windows	MLWE
13		–Modify location of a widget to overlap with another	MLWO
14		–Modify size of widgets	MWS
15		–Modify appearance of widgets	MWA
16		–Modify type of widgets (Button changed to TextField)	MWT
17		–Modify GUI library for widgets (Swing button changed to AWT Button)	MWL
18		–Expand/Reduce size of Windows and Widgets will Adjust their Sizes	EWWAR/ RWWAS

MOs belonging to the "Adding" class imply that functionality is added to the SUT to change its behavior. For instance that an additional conditional statement is added to a method or class. The "Modifying" class implies that functionality is changed to change the SUT's behavior. For instance that the statement of conditional statement is changed from one operator to another, e.g. "less than" to "less than equal" or "greater than". Finally, "Removing" implies the removal of functionality from the SUT. For instance, a conditional statement is removed from a method or class. These operators have, as stated, been implemented in tools that can automatically mutate a SUT.

B. Concepts on generic MOs for GUI-level

The operators presented in Table I are derived from the generic operators defined in the body of knowledge on mutation analysis and provide a comprehensive list of 18 operators to create GUI-level mutants. Below, divided by classes, we present an analysis of each operator, providing examples of how they can be applied in practice. One can notice that the provided examples are not comprehensive, which means it is possible to create many different instances of each operator, similarly to some traditional mutation operators. In the context of this study, "widget" is defined as a component on the SUT's GUI that can receive input or display output, e.g. buttons, text fields, canvases, etc.

1) Removing MOs: Remove Existing Widget (REW): This operator includes all instances where a widget or widgets are removed from the SUT's GUI. This does not imply that the underlying functionality associated with the widget is removed. The operator asserts that the test suite fails given that expected widget is not visible on the screen or in the GUI model.

Set Widget Invisible (SWI): In this instance the widget is removed from the pictorial GUI but is still connected to the underlying functionality and visible in the GUI model. The purpose of the operator is to verify that the test suite fails if the instance occurs. For GUI-driven techniques that interact with the GUI-model this could imply a requirement to verify that the widget's opacity/visibility properties are enabled.

Remove Existing Listener (REL): This operator aims at evaluating the test suite/techniques ability to correctly assert the outcome of an interaction. Hence, it asserts the test's ability to fail when SUT stimuli does not result in any GUI output.

2) Adding MOs: Add Identical Widget (AIW): This MO asserts that the test technique/suite can uniquely identify a widget regardless of its properties and appearance. This instance is valid for GUI's with many equal widgets that are by a human uniquely identified by another widget, e.g. a toggle button next to a labeled pane. Failure to identify this mutant can imply that the search properties of the test technique require modification.

Add Similar Widget (ASW): This operator serves the same purpose as AIW. However, the number of properties between the different widgets differ more than for AIW.

Add Different Widget (ADW): In this operator a new widget is added to the GUI, e.g. a button or window. The operator asserts that changes to the GUI does not affect the test suite. Instances where it can fail is when the new component changes in the property information of other widgets, e.g. ID numbers, or moves a widget to a new location where it is no longer visible.

Add Another Listener (AAL): Similar to operator REL, this operator asserts that the test technique/suite can identify a discrepancy between the actual and expected output. The operator can be implemented by connecting the same listener twice to a widget or by connecting a listener from another widget to the mutated widget.

3) Modifying MOs: Expand/Reduce size of Windows and Widgets Will Auto-adjust their Sizes (EWWAR/ RWWAR): This operator asserts that the test technique/suite behaves correctly independent of widget size, resolution, etc. The operator also

implies that the SUT's GUI widgets are scalable. This operator is of particular importance for mobile applications that should support different mobile devices in different orientations, and operating systems.

Reduce size of Windows to Hide Widgets (RWHW): Similar to the operators REV and SWI, this operator asserts that the test technique/suite behaves correctly if widgets are hidden from view. Note that this operator still implies that the widgets are visible in the GUI model.

Modify Location of a Widget to a Proper location (MLWP): In this instance the operator asserts that the test suite behaves correctly even if the widgets' locations are changed. The simplest implementation of this operator is to switch the location of one or several widgets. This operator does however still require the widget to be moved to another location that is visible on the screen.

Modify Location of a Widget to Edges of windows (MLWE): Unlike MLWP, this operator requires that the mutated widget is only partially visible on the SUT's GUI. Note that this can cause problems for GUI model driven approaches since the widget is still fully visible in the GUI model.

Modify location of a widget to overlap with another (MLWO): Similar to MLWE this operator implies that one, or several, widgets are only partially visible on the pictorial GUI. The operator asserts the test technique's sensitivity to GUI layout.

Modify Size of Widgets (MWS): Similarly to EWWAR/RWWAR, this operator asserts that the testing technique/suite behaves correctly when the widgets' resolution or size changes. Note that a correct behavior is dependent on if the change of the widget is intentional or unintentional. An intentional change implies that the test should pass and an unintentional change that it should fail.

Modify Appearance of Widgets (MWA): Similar to MWS this operator asserts the behavior of the test technique/suite based on the intentional/unintentional change of a widget. It implies that the test should *fail* if the change is unintentional and *pass* if it is intentional. Changes to the appearance can be implemented by changing color, font and shape of the widget. Note that size is covered by MWS and WWAR/RWWAR.

Modify Type of Widgets (Button changed to TextField) (MWT): This operator asserts the test technique's/suite's ability to distinguish between different widget types. This is especially important for SUT's with custom widget since changes to the widgets should not affect the test suite's robustness.

Modify GUI Library for Widgets (Swing button changed to AWT Button) (MWL): This operator asserts that the testing technique/suite is applicable on SUT's built with several different GUI libraries or in system's where the GUI libraries are interchangeable. The expected outcome of the test in this instance is that it should pass the test case after the GUI library has been changed.

IV. Methodology

We now provide a detailed view of our research questions, the methodology we followed, and the experimental strategies we have designed. Basically, this study aims at assessing the fault-generation effectiveness of a set of GUI-based MOs and a comparative analysis between: (1) traditional MOs, and (2) GUI-based MOs for GUI-level mutation analysis.

A. Research Questions

The study is designed to answer the following *Research Questions* (RQ):

RQ1: Are the defined GUI-based MOs more effective on seeding faults associated to the GUI level than traditional MOs?

RQ2: Is it possible to automate the generation of GUI mutants?

RQ3: Is it possible to use these proposed GUI operators on a real-world GUI application?

B. Research Strategy

To find empirical evidences to answer the aforementioned RQs, we divided our study into three parts:

(1) A pilot experiment, in which we have applied a semi-automatic strategy to generate mutants using a set of random GUI-based MOs; We first explored a set of nine mutation operators in a simple Swing GUI-based application. In this part of the study, we also identified generic ways to automate generate mutants, through scripts, for applications developed with the Java GUI library Swing. In addition, in this pilot study we have included a comparison between traditional method-level mutation operators generated by MuJava [9] and a set of specific GUI-based MOs;

(2) A proof-of-concept study with an OSS GUI-based application. More specifically, a randomly selected GUI of WEKA [10], a tool with a collection of machine learning algorithms for data mining tasks. The study was performed by applying a semi-automated strategy to generate mutants to observe the operators, and the strategies, feasibility in a real-complex environment. A semi-automated approach was required since some operators proved challenging to automate, which we return to in Section V-A2; and

(3) An analysis of some important issues connected to GUI testing, detailed in Section IV-E. This analysis aims at evaluating the effect of the mutation operators on the GUI, we have analyzed. In addition, to measure how useful the GUI-based mutation operators are, this part of our study aimed at identifying issues of equivalence and redundancy among the current mutants generated from our approach.

C. Pilot experiment: Java Swing Calculator

In order to provide research data towards answering RQ1 and RQ2, in our pilot study we have followed a sixfold strategy: (1) choosing as subject application a random GUI-based application from an open-source code repository; (2) randomly select three mutation operators from each class of GUI-based MOs (Table I); (3) analyzing the source code of the subject application to identify strategies of how to automate the generation of the mutants through the MOs selected in step two; (4) generating, automatically or manually, a reasonable

amount of mutants for each GUI-based mutation operator we have selected; (5) using MuJava to generate traditional method-level mutants for the same application; (6) manually analyzing the results of the MOs regarding their GUI-level effects and how these effects impact different GUI-level testing technologies.

This pilot experiment was designed mainly with two objects: (1) using a sample of the GUI-based mutation operators, evaluate their applicability and measure their efficacy;

(2) compare the GUI effect of our mutation operators and traditional mutation operators.

1) Subject application: We selected a GUI-based calculator application implemented in Java with the GUI library Swing – in the continuation of the paper referred to as "*Java Calculator*". The choice of this application is based on two reasons: (1) simplicity of the application; and (2) well-defined functionality to be performed through an intuitive GUI.

Figure 1 presents the GUI of *Java Calculator*.

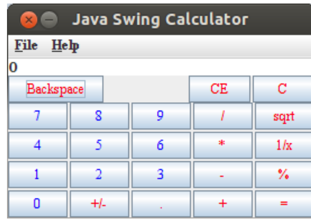
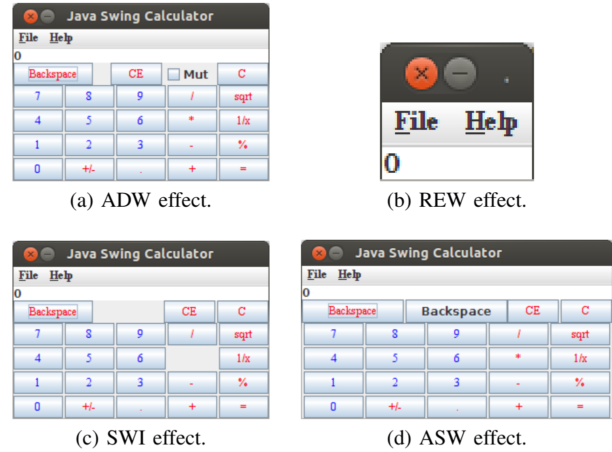


Fig. 1: Original GUI of *Java Calculator*.

2) Applying the GUI-based mutation operators: We have randomly selected three different GUI-based MOs from each class. Specifically, the mutation operators accronymed REW, SWI, and REL were selected to represent the "removing" class. In turn, AIW, ASW, and ADW were selected to represent the "adding" class. Finally, MWS, EWWAR-RWWAR, and RWHW were chosen to represent the "modifying" class. We have selected nine GUI mutation operators because MuJava implements nine traditional mutation operators on a method level. Then, to starting establishing a line of comparison, we decided to use the same number of GUI-based MOs.

After studying each one of the mutation operators and the original source code of the subject application, we have implemented several scripts to support the automatic generation of GUI mutants. Out of the nine randomly selected mutation operators, we created automated scripts for seven of them. Due to their complexity, mutants generated from the MOs called EWWAR/RWWAR and RWHW were generated manually.

At the end of the mutation process, 87 mutants were obtained with several GUI effects. As many of the mutants as possible were generated automatically. Regarding the mutants generated manually that we could not automate, we generated at least one mutant to represent the visual appearance of the potential error according to the specification of the MO. Figure 2 presents four GUIs of muted versions of *Java Calculator* generated from different MOs. We took several screenshots of all of the mutants generated by our approach that can be accessed online[3].



(a) ADW effect. (b) REW effect. (c) SWI effect. (d) ASW effect.

Fig. 2: Four mutated instances of the *Java Calculator*.

After generating the mutants, we analyzed their effect on the application's GUI. We have had a special attention in some particular aspects detailed in the third part of our research strategy (Section IV-A). All of our findings are presented and discussed in Section V.

3) Study on applying traditional MuJava MOs: MuJava implements 12 method-level mutant operators for Java programs. Table II details all of the traditional MuJava MOs, originally proposed by [11]. Some of the MOs may have slight variations such as, AOI (*Arithmetic Operator Insertion*) that can be implemented through basic unary operators (AOI_U) or short-cut operators (AOI_S).

TABLE II: Tradition method-level MOs implemented by MuJava

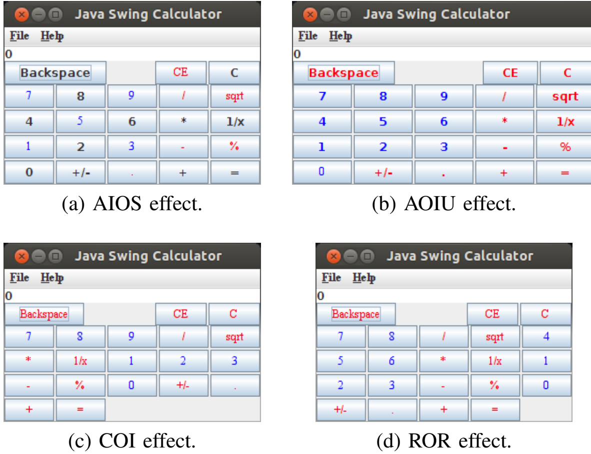
MuJava – Traditional method-level MOs		
Operator	**Description**	**Variations**
AOR	Arithmetic Operator Replacement	AOR_B, AOR_U, and AOR_S
AOI	Arithmetic Operator Insertion	AOI_U and AOI_S
AOD	Arithmetic Operator Deletion	AOD_U and AOD_S
ROR	Relational Operator Replacement	–
COR	Conditional Operator Replacement	–
COI	Conditional Operator Insertion	–
COD	Conditional Operator Deletion	–
SOR	Shift Operator Replacement	–
LOR	Logical Operator Replacement	–
LOI	Logical Operator Insertion	–
LOD	Logical Operator Deletion	–
ASR	Assignment Operator Replacement	–

408 mutants were generated by MuJava that have, like previous mutation results, been posted online[4]. Despite we used all of the available method-level mutation operators, we observed that a set of nine mutation operators out of 12 (plus its variations) actually generated mutants: AOI_S, AOI_U, AOR_S, *COI*, *LOI*, *ROR*, AOR_B, *COD*, and *COR*. Then, *AOD*, *SOW*, *LOR*, *LOD*, and *ASR* did not generated any mutant. Figure

[3]All of the data collect from this pilot experimentation is available. See: http://www.labes.icmc.usp.br/∼rpaes/ICST2015Data/Mutation2015.html

[4]All of the data collect is available online. See: http://www.labes.icmc.usp.br/∼rpaes/ICST2015Data/Mutation2015-2.html

3 presents some examples of the effect of traditional MOs on the *Java Calculator*'s GUI. After the mutants had been created they were all visually analyzed to identify what effect the method level operators had on the GUI. The analysis is presented in more detail in Section V.



(a) AIOS effect. (b) AOIU effect.

(c) COI effect. (d) ROR effect.

Fig. 3: Mutants of *Java Calculator* generated from traditional MuJava MOs.

D. Proof-of-concept: real-world complex GUI-based OSS

To answer RQ3, we conducted a proof of concept study where the nine MOs from part 1 of the study were applied on a larger OOS application – WEKA. The study was performed in four steps: (1) selecting a real-complex Java Swing application from web; (2) studying its GUI code and selecting one of its GUIs to apply the set of nine GUI-based MOs explored during the pilot (Section IV-C); (3) generating the mutants through the previous implemented scripts or manually when there is no script available; and (4) manually analyze the resulting mutants regarding their GUI effects and implications for testing with different GUI-based test technologies and regarding the step three of our research strategy.

Differently from the pilot experiment, this proof-of-concept study had only one goal: provide support for the applicability of the GUI-based mutation operators in real GUI applications.

1) Subject application: WEKA[5] is a well known environment that supports users to solve data mining problems through a set of machine learning algorithms [10]. WEKA, which is an open-source application implemented under the *General Public License version 3.0* (GPLv3), includes a set of user interfaces implemented using Swing. Our choice to explore this application is because WEKA's source code is organized in more than 80 packages and its GUI code is easy to understand and manipulate. Thus making the application notably representative of real-world software but yet easy to manipulate as required in our study.

Table III presents some metrics[6] we have collected from *Java Calculator* in comparison to WEKA's source code. The differences between the projects are notable. Despite this fact, after manual analysis, we have noticed the WEKA's GUI code is well documented and understandable.

TABLE III: Complexity analysis of *Java Calculator*

Metrics from the subject applications					
App.	**NoC**	**NoM**	**NoSM**	**LoC**	**CC**
Java Calculator	3	19	1	461	4.55
WEKA	1230	16570	1241	260226	2.82

NoC – Num. of Classes; **NoM** – Num. of Methods
NoSM – Num. of Static Methods; **LoC** – Lines of Code
CC – Cyclomatic Complexity (average)

After analyzing WEKA's source code, we identified a part of WEKA's GUI to apply our GUI-based MOs on. Figure 4 represents the visual appearance of the WEKA's GUI for pre-processing dataset useful for data mining experiments. One can see that the GUI is simple and it includes a lot of different Swing components.

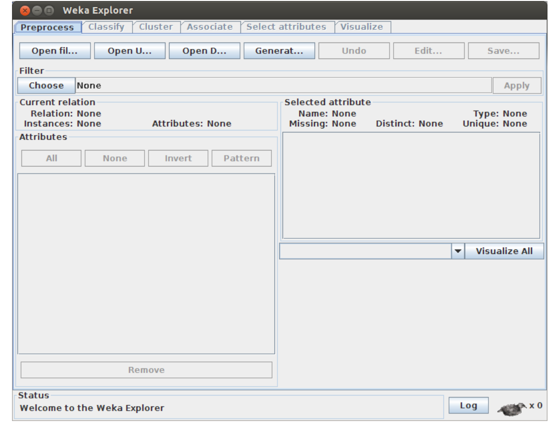


Fig. 4: WEKA's original pre-processing GUI.

2) GUI-based MOs applied on WEKA: Aiming to generate mutants using the GUI-based MOs, we have set our previously implemented scripts to work properly on the source code of WEKA's pre-processing GUI. For some MOs, we had to set the scripts with some exceptions treatments to avoid generating invalid mutants. This process was done manually, however, overall, the process of generating mutants through scripts, or manually, was quick. Hence, the human efforts required to generate mutants were alleviated by the scripts.

The process of generating mutants for WEKA has resulted in 130 versions of the original code. A deep analysis on all of the mutated WEKA's GUI is available online[7]. Figure 5 shows the visual appearance of a mutated WEKA's pre-processing GUI. Section V presents a wide discussion on the applicability of the GUI-based MOs in real complex GUI-based applications. All of the GUI effects are deeply analyzed to support the discussions and findings presented in this paper.

[5] See more about WEKA: https://sourceforge.net/projects/weka/

[6] The Eclipse plug-in's "Metrics" was used to measure the code metrics: http://metrics.sourceforge.net/

[7] Data collect from proof-of-concept and pilot study are available online. See: http://goo.gl/T8JMym

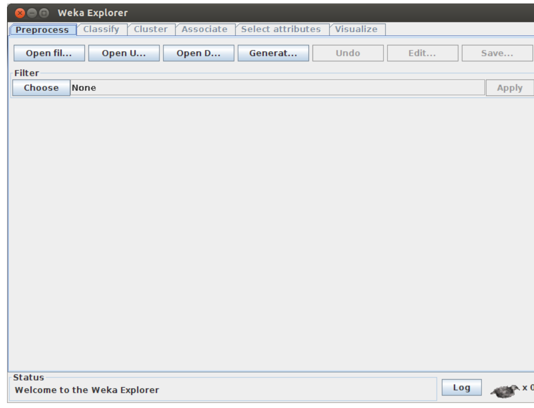


Fig. 5: Mutant of *WEKA* generated from GUI-based MOs.

E. Quantitative analysis and of evaluation of properties of MOs for the GUI-level

For both experimentation scenarios (pilot and real-word application) we have analyzed different aspects: numbers of mutants generated, visual equivalent mutants, structural equivalent mutants, capacity of generating mutants, etc. From these numbers, we performed a quantitative and qualitative analysis among the MOs involved in our experiment.

In addition to that, we formulated a yes/no questionnaire to identify properties associated with the GUI generated from the MOs. This questionnaire aims to identifying properties associated with each MOs regarding both scenarios "pilot" and "real-world application". Table IV presents these properties we defined. These properties are important for GUI testers, once they can differ depending on the generation of the GUI-testing approach.

TABLE IV: Questionnaire – Properties of GUI-based MOs.

	Yes/No – GUI-based MOs
P1	Does it change the GUI appearance?
P2	Does it change the GUI model/structure in testing tools?
P3	Can it change the SUT's input behavior?
P4	Can it change the SUT's output behavior
P5	Does it create equivalent mutants?
	P5.1 – Structural equivalent mutants?
	P5.2 – Visual equivalent mutants?

V. Result Discussions

The results collected from the empirical study detailed in Section IV. In Section V-A we present potential answers to the RQs. Based on our findings, Section V-B presents several benefits for GUI testers on using GUI-level mutation operators. Further, in Section V-C, based on our own point of view, we elicit future directions for the concepts of mutation analysis and GUI-level testing.

A. Answers to the RQs

Regarding the previous defined RQs (Section IV-A) our experiments contribute to clarify several points. Based on the empirical data we have collect, below we present all of the evidences that contribute to respond the RQs.

1) RQ1: Our empirical data and the study conducted reveal that regarding the comparison among traditional MOs and GUI-based MOs, the second approach is not only more effective on seeding faults associated to the GUI level, but also it generates mutants able to represent more complete set of GUI-associated faults. This is due to the fact of GUI-based MOs were specifically designed to reproduce different faults associated to the GUI-level of the SUT. In addition, using GUI-based MOs can be more productive for GUI testers, once a reduced set of mutants is generated to represent a complete set of potential GUI faults.

Supporting data to answer this question can be found in Table V, which presents a comparison among nine traditional MOs and nine GUI-level MOs. For *Java Calculator* the table presents an analysis of more than 400 mutants generated with traditional MOs and 87 GUI-based MOs, which were generated with a semi-automated approach. All of the mutants were manually analyzed and classified considering three issues:

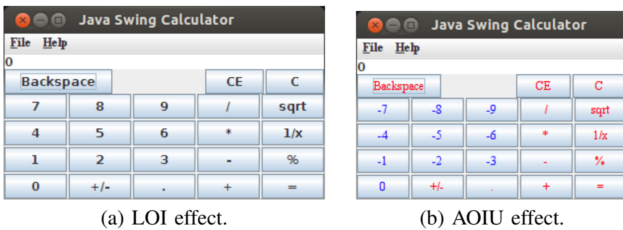
- Alive vs Dead: there are some cases in which the mutated code is not possible to run due to the throw of some unpredictable exception. These cases were common for mutants generated with traditional MOs. For the scope of our study, when a mutated code throws an exception and the GUI can not be available due to this, we consider the mutant as "dead", otherwise, the mutant is considered "alive";
- GUI effects: observing the GUI of the subject application, for both approaches, we have noticed some MOs reflect directly on the GUI, modifying its visual appearance. Then, regarding the two mutation approaches (traditional and GUI-based) we only counted the number of mutants with effects on the GUI;
- Mutant Equivalence: for GUI testing, we have grouped equivalence into two groups: (1) visually equivalent; and (2) GUI-model equivalent. Visual equivalent is a mutant that generates a visually identical GUI in comparison to the original GUI of the SUT. GUI-model equivalent is a mutant that generates an event-flow graph identical to the graph generated for the original GUI. Table V presents only the visually equivalent mutants. However, we have analyzed all of the mutants also for GUI model equivalence. Two of the authors of this paper have conducted these processes manually; and
- Redundancy: In both approaches, we have noticed several mutants, that were generated from the same MOs, are identical from a GUI testing context. For example, depending on the GUI Layout employed, SWI and REW may generate visually identical mutants. In the scope of this study, exploring concepts of previous investigations [12], we called this mutants redundant. In this context, our study performed an analysis of each MOs aiming to identify these particular cases.

Table V, aforementioned, presents the results from the mutation analysis strategy applied on the application *Java Calculator* using traditional MOs and GUI-based mutations

operators. For each MO, the table shows the total number of mutants, the number of invalid/dead mutants, the number of alive mutants with an effective GUI effect, the number of visually equivalent mutants, and, finally, some statistics on the efficiency of the MO regarding the GUI testing.

Traditional MOs presented a mean of 11.27% of GUI-fault seeding effect, i.e. useful GUI mutants. In comparison, GUI-based MOs presented a mean of 83.90% of GUI-fault seeding effect, implying their proficiency for GUI testing experiments over method-level MOs. In addition, traditional mutation operators generate a huge number of equivalent mutants (73%), which implies low productivity of usable/valuable GUI mutants.

Analysis of the GUI mutants created by the traditional MOs also showed that all mutants were either of type MWA (*Modify Appearance of Widgets*) or MLWO (*Modify the Location of a Widget to Overlap Another*). Figure 6 shows an example of one of these cases. This is because the mutants generated by traditional MOs are primarily only able to change font size, font color (Fig. 6a), widget text (Fig. 6b), etc. The other reason was because the mutated application contained a vector of buttons, which order could be changed by the traditional operators.



(a) LOI effect.

(b) AOIU effect.

Fig. 6: Effect of traditional MOs on the GUI-level.

2) RQ2: The GUI-based MOs have different goals in comparison to traditional MOs, therefore the answer to RQ2 is based on empirical support gained by the automation of GUI-level mutant generation.

In our empirical analysis, we implemented "semi-automated" scripts to generate mutants for seven different MOs for the Java Swing GUI library. These scripts are referred to as semi-automated because they do not execute the mutant version of the code. The scripts are able to manipulate the original source-code, looking for potential commands in which the mutation reflects on the GUI. The step-by-step of this approach is: searching for specific keywords (depending on the MO), modifying the original code, and saving the "mutant" in a specific folder with a suggestive name.

Technically, each script works following a five-steps work-flow: (1) read the original code as a text file; (2) find some target command or specific keyword on the code; (3) replace, comment, or rejoin with other pre-defined code; (4) save the modified version of the code in a new folder; and (5) repeat steps 2 to 5 until reach the end of the file.

Regarding this framework to automatically generate GUI-based mutants for Swing applications, one can notice that each MO may be implemented manipulating some specific methods or functions provided by the GUI library. For instance, the mutant operator REW, which removes an existing widget, could be implemented through the omission of the method `public Component add(Component comp)`, which adds a widget in a pre-defined component. We have implemented this omission through a script that is able to identify and comment all of the "add" commands in the original code. Similarly, the mutation operator SWI is associated with the method `public void setVisible(boolean aFlag)` implemented by the Swing class called `JComponent`. In this particular case, we have implemented a script to add a command `setVisible(false)` in all of the pre-defined components in the source code. Despite our well-succeed usage of this framework, we predict that the use of variate Layouts may bring defective mutants.

Regarding the nine GUI-based MOs involved in our study, we consider the automation of two of them (EWWAR/RWWAR and RWHW) as a non-trivial task. Due to this, we created their mutants manually. This is due to the complexity associated with the task of automatically identifying visual content on the current GUI of the application. These hard-to-automate MOs are directly associated with cognitive tasks of human beings such as, resizing a GUI until hiding a component. Then, these two MOs are intimately associated to the size of the screen, fonts, and the sizes of the widgets inside the GUI. In addition, their complete automation requires the association of computer vision techniques. Furthermore, the complexity of the automation of these operators come from dependencies from different places in the source code that must all be changed to get the desired GUI level defect. Further, because these changes need to be done on variables with potentially unknown variable names it is difficult to create a generic script that implements these MOs. We conclude that a more sophisticated automation strategy is necessary such as, a strategy using the resources of Java Virtual Machine.

Through our study it is possible to affirm that the automatic generation of GUI-based mutants is feasible for most of the defined MOs. However, different strategies must be followed for different GUI libraries. To support this, after studying particular characteristics of the Swing library, we followed a simple work-flow we developed, enabling the automation of generation of mutants for seven MOs. Our efforts on developing and testing all of these scripts were about three hours of coding. Once the tester knows the GUI library, this cost of implementation tends to decrease.

3) RQ3: The GUI-based MOs behaved properly in a real-world complex GUI-based application. Manually and through the previously implemented scripts, 87 mutants were generated for a single GUI of subject application.

We had no previous contact with WEKA and after quickly analyzing its source code, we selected an aleatory screen and started applying the scripts right away. WEKA has a highly documented source-code and its GUI code is well organized through packages with suggestive names.

Regarding this research question, we highlight some cases

TABLE V: *Java Calculator* – Traditional vs GUI-based mutation operators

Statistics on using traditional mutation operators								
trad. MO	**# mut.**	**# dead/invalid**	**# alive**	**# alive + no GUI eff.**	**# alive + GUI eff.**	**# equiv.**	**“Good” Mut.**	**efficiency (%)**
AOIS	183	17	166	17	149	135	14	**7.65**
AOIU	33	6	27	0	27	22	5	**15.15**
AORS	6	5	1	0	1	1	0	**0.00**
COI	25	1	24	0	24	19	5	**20.00**
LOI	44	11	33	0	33	26	7	**15.91**
ROR	92	5	87	0	87	72	15	**16.30**
AORB	16	0	16	0	16	16	0	**0.00**
COD	3	0	3	0	3	3	0	**0.00**
COR	6	0	6	0	6	6	0	**0.00**
total	408	45	363	17	346	300 (73%)	46	**11.27**
Statistics on using GUI-based mutation operators								
trad. MO	**# mut.**	**# dead/invalid**	**# alive**	**# alive + no GUI eff.**	**# alive + GUI eff.**	**# equiv.**	**“Good” Mut.**	**efficiency (%)**
REW	13	0	13	0	13	0	13	**100**
SWI	11	0	11	0	11	0	11	**100**
REL	13	0	13	0	13	13	0	**0.00**
AIW	11	0	11	0	11	0	11	**100**
ASW	11	0	11	0	11	0	11	**100**
ADW	11	0	11	0	11	0	11	**100**
MWS	10	0	10	0	10	0	10	**100**
RWHW	5	0	4	0	4	0	4	**80**
EWWAR / RWWAR	2	0	2	0	2	0	2	**100**
total	87	0	86	0	86	13 (14.94%)	73	**83.90**

in which several equivalent mutants were generated. In percentage, equivalent mutants were more representative than in the pilot study. Analyzing the WEKA’s GUI source code, we noticed this is due to several secondary GUI screens such as confirmation dialog were implemented inside the listener of widgets of the mutated GUI.

To find support for the applicability of the GUI-based MOs in real-world applications, Table VI presents an analysis of five important properties for GUI-testing collected from the yes/no questionnaire previously detailed in Table IV. These question are important in the context of they may influence on the testing activities depending on the generation of the GUI-testing tool being used. Table VI is divided into two different perspectives: pilot study and real-world applications. The goals of the table is to show if the properties are the same pilot study and the proof-of-concept study. Then, we highlight that the properties were kept in more than 90% of the cases, providing support of the applicability of the GUI-based mutants in complex scenarios. In some special cases, the property can be different, depending on the generation of the GUI-testing tool used, in these particular cases, the cell in the table was set as “y/n”.

B. Benefits on using GUI-based MOs

The main benefit of using GUI-based MOs, according to the empirical data produced from this study, is the fact that they generate mutants that reflect directly on the GUI-level of the SUT. Additionally, traditional MOs tend to generate several visual equivalent mutants, making the testing activities on the GUI-level costly. Equally, traditional MOs generate a good amount of mutants that throw unpredictable exceptions, precluding the mutation analysis. Finally, we have empirically demonstrated that mutants from traditional MOs are able to reproduce only few types of GUI-level faults. In this sense, using GUI-based MOs assures testers are dealing with a more complete and representative set of faulty GUIs.

Despite our results, we highlight that system testing at the GUI level is equally important and valuable as functional testing, and the GUI mutants we propose here are useful at this level and outperforms traditional mutants. However, regarding the system as a whole, we consider both type of mutants helpful for testing activities, increasing the SUT realizability.

C. Mutation analysis and GUI testing: Future directions

The application of mutation analysis concepts, in the context of GUI testing, is an open research field. As soon as the GUI-level testing has been drawing the attention of researchers and practitioners, its association with a powerful test strategy like mutation testing can be seen as a predictable course. All open research fields suffer from the lack of automation support and the need for standardization of approaches and strategies. In this context, it is necessary to implement effective tools to support the GUI-level mutation analysis. Most of this paper contributes towards the definition of a tool to this purpose. Then, regardless the GUI library, the main future direction we see for this area is the development of a complete tool aiming to alleviate human costs on selecting MOs, generating mutants, loading testing data, taking screenshots, marking equivalent mutants, etc.

VI. Threats to Validity

Below we present present the threats of this study on five different perspectives:

Internal validity: The presented, multi-step, empirical study was exploratory in nature but also focused with a narrow scope, driven by a set of well defined research questions and

TABLE VI: Properties' analysis: Pilot (*Java Calculator*) VS Real-World application (WEKA)

	Remove						Duplicate/Instantiate						Modify					
	REW		SWI		REL		AIW		ASW		ADW		MAWS		EWWAR/ RWWAR		RWHW	
#	Pil.	R.W	Pil.	R.W	Pil.	R.W	Pil.	R.W	Pil.	R.W	Pil.	R.W	Pil.	R.W	Pil.	R.W	Pil.	R.W
P1	yes	y/n	yes	yes	no	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
P2	yes	yes	no	no	no	no	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no	no
P3	yes	yes	yes	yes	no	no	no	no	yes	yes	no	no	yes	yes	yes	yes	y/n	y/n
P4	yes	yes	yes	yes	yes	yes	no	no	no	no	no	no	no	no	no	no	y/n	y/n
P5	no	yes	yes	yes	yes	yes	no	yes	no	yes	no	y/n	yes	yes	yes	yes	yes	yes
P5.1	no	yes	no	yes	no	no	–	no	–	yes	–	–	yes	yes	yes	yes	yes	yes
P5.2	yes	yes	yes	yes	yes	yes	–	yes	–	yes	–	–	yes	yes	yes	yes	yes	yes

P_n – Property n, see Table IV
Pil. – Result collected from the experiment involving the pilot
R.W – Result collected from the experiment involving the Real-World application

properties. The results of the study provide support to answer the research questions and the results between steps were comprehensively consistent. As such the internal validity of this study is considered high.

External validity: The study evaluates the effectiveness of the GUI-based and traditional MOs on both a small calculator application and a larger OOS application. However, as the OOS application was still small compared to industrial software, we can only claim moderate external validity and pose that further research is required to validate our results on more systems and platforms. Further work is also required since our study was restricted to Java Swing based applications.

Construct validity: As the concept of mutation is general to all applications, the construct validity in this case is considered high. Hence, the size, and complexity, of the chosen applications are suitable to show the operators effectiveness for Java based applications.

Conclusion validity: We have presented the methodology in full and posted all study results, i.e. screenshots, online for the reader to view. These results support our conclusions and we therefore claim that we have high conclusion validity.

VII. Conclusion

This paper presents an empirical analysis of the fault seeding-effectiveness of generic GUI-based MOs. These MOs are useful to measure the efficiency of new GUI-based testing strategies, once the mutants generated from them reflects on the GUI-level, simulating faulty GUIs. Based on three research questions, we conducted empirical analysis on two different applications: (1) a regular *Java Calculator*, and (2) a real-world complex application. Aiming to help GUI-based testers, our findings showed the efficiency of using GUI-based MOs to generate faulty GUIs, even if the tester is dealing with a real-world complex GUI-based application. In addition, empirical data demonstrates that traditional mutation operators tend to be useless for GUI testing. However, we highlight that for the SUT as whole, regarding underline code testing activities, traditional MOs are fundamental and their efficiency is unchallengeable.

Through a deep analysis on the number of useful mutants for GUI testing, our study reveals that the GUI MOs had almost 84% of effectiveness, in contrast to only 12% of the traditional method-level approaches. Finally, our study demonstrates that is possible to implement supporting tools to automate the GUI-based MOs fault-seeding process, being in many cases a non-trivial task. A supporting framework to automate GUI-based MOs for Swing application is provided. However, in some cases, due to widget dependencies in the code it is challenging to create an automated script. For future work, we propose the creation a generic framework of activities to automate the MOs for different GUI libraries and platforms. In addition to our findings, this framework opens channels for technology transfer, alleviating human efforts and completing the set of contribution of the research reported in this paper.

References

[1] A. Memon, M. Pollack, and M. Soffa, "Hierarchical gui test case generation using automated planning," *IEEE Trans. on Sof. Eng.*, vol. 27, no. 2, pp. 144–155, Feb 2001.

[2] X. Yuan, M. Cohen, and A. Memon, "Gui interaction testing: Incorporating event context," *IEEE Trans. on Sof. Eng.*, vol. 37, no. 4, pp. 559–574, July 2011.

[3] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (gui) testing: Systematic mapping and repository," *Inf. and Soft. Techn.*, vol. 55, no. 10, pp. 1679–1694, 2013.

[4] E. Alégroth, "On the industrial applicability of visual gui testing," Department of Computer Science and Engineering, Software Engineering (Chalmers), Chalmers University of Technology, Goteborg, Tech. Rep., 2013.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–43, 1978.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. on Sof. Eng.*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[7] Q. Xie and A. M. Memon, "Using a pilot study to derive a gui model for automated testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 7:1–7:35, Nov. 2008.

[8] E. Alégroth, Z. Gao, R. A. Oliveira, and A. Memon, "Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study," in *ICST 2015*, 2015, pp. n/a–n/a, to appear.

[9] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009.

[11] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An experimental mutation system for java," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–4, 2004.

[12] C. Wright, G. Kapfhammer, and P. McMinn, "The impact of equivalent, redundant and quasi mutants on database schema mutation analysis," in *QSIC 2014*, Oct 2014, pp. 57–66.