Classifying and Qualifying GUI Defects

Valéria Lelli, Arnaud Blouin, Benoit Baudry

To cite this version:

Valéria Lelli, Arnaud Blouin, Benoit Baudry. Classifying and Qualifying GUI Defects. 8th IEEE International Conference on Software Testing, Verification and Validation, Apr 2015, Graz, Austria. pp.1-10, 10.1109/ICST.2015.7102582 . hal-01114724v2

HAL Id: hal-01114724
https://hal.inria.fr/hal-01114724v2

Submitted on 9 Jul 2015

Classifying and Qualifying GUI Defects

Valéria Lelli
INSA Rennes, France
valeria.lelli_leitao_dantas@inria.fr

Arnaud Blouin
INSA Rennes, France
arnaud.blouin@irisa.fr

Benoit Baudry
Inria, France
benoit.baudry@inria.fr

***Abstract*—Graphical user interfaces (GUIs) are integral parts of software systems that require interactions from their users. Software testers have paid special attention to GUI testing in the last decade, and have devised techniques that are effective in finding several kinds of GUI errors. However, the introduction of new types of interactions in GUIs presents new kinds of errors that are not targeted by current testing techniques. We believe that to advance GUI testing, the community needs a comprehensive and high level GUI fault model, which incorporates all types of interactions. The work detailed in this paper establishes 4 contributions: 1) A GUI fault model designed to identify and classify GUI faults. 2) An empirical analysis for assessing the relevance of the proposed fault model against failures found in real GUIs. 3) An empirical assessment of two GUI testing tools (*i.e.* GUITAR and Jubula) against those failures. 4) GUI mutants we've developed according to our fault model. These mutants are freely available and can be reused by developers for benchmarking their GUI testing tools.**

I. Introduction

Increasing presence of system interactivity requires software testing to closely consider the testing of graphical user interfaces (GUI). GUIs are composed of graphical objects called widgets, such as buttons. Users interact with these widgets (*e.g.* press a button) to produce an action[1] that modifies the state of the system. For example, pressing the button "*Delete*" of a drawing editor produces an action that deletes the selected shapes from the drawing. Most of these standard widgets provide users with an interaction composed of a single input event (*e.g.* pressing a button). In this paper we call such interactions "mono-event interactions". These standard widgets work identically in many GUI platforms. In the context of GUI testing, the tools rely on the concept of standard widgets and have demonstrated their ability for finding several kinds of errors in GUIs composed of such widgets, called WIMP[2] GUIs [3], [4], [5], [6], [7].

The current trend in GUI design is the shift from designing GUIs composed of standard widgets to designing GUIs relying on more complex interactions[3] and *ad hoc* widgets [2], [8], [9]. So, standard widgets are being more and more replaced by *ad hoc* ones. By *ad hoc* widgets we mean non-standard widgets developed specifically for a GUI. Such widgets involve multi-event interactions (in opposition to mono-event interactions, *e.g.* multi-touch interactions for zooming, rotating) that aim at being more adapted, natural to users. A simple example of such widgets is the drawing area of graphical editors with which users interact using more complex interactions such as pencil-based or multi-touch interactions. GUIs containing such widgets are called post-WIMP GUIs [10]. The essential objective is the advent of GUIs providing users with more adapted and natural interactions, and the support of new input devices such as multi-touch screens. As Beaudouin-Lafon wrote in 2004, "*the only way to significantly improve user interfaces is to shift the research focus from designing interfaces to designing interaction*" [8]. This new trend of GUI design presents to developers new problems of GUI faults that current GUI testing tools cannot detect. An essential pre-requisite to propose comprehensive testing techniques for both WIMP and post-WIMP GUIs is to define an exhaustive and high level GUI fault model. Indeed, testing consists of looking for errors in a program. This requires a clear idea about the errors we are looking for. This is the goal of fault models that permit to qualify the effectiveness of testing techniques [11].

In this paper, we leverage of the evolution of the current Human-Computer Interaction (HCI) state-of-the-art concepts to propose an original, complete fault model for GUIs. This model tackles dual objectives: 1) provide a conceptual framework against which GUI testers can evaluate their tool or technique; and 2) build a set of benchmark mutations to evaluate the ability of GUI testing tools to detect failures for both WIMP and post-WIMP GUIs. We assess the coverage of the proposed model through an empirical analysis: 279 GUI-related bug reports of highly interactive open-source GUIs have been successfully classified using our fault model. Also, we assess the ability of two GUI testing tools (*i.e.* GUITAR and Jubula) to find real GUI failures. Then, from an open-source system we created mutants implementing the faults described in our fault model. These mutants are freely available and can be used for benchmarking GUI testing tools. As an illustrative use of these mutants, we conducted an experiment to evaluate the ability of two GUI testing tools to detect these mutants. We show that some mutants cannot be detected by current GUI testing tools and discuss future work to address the new kinds of GUI faults.

The paper is organized as follows. The next section examines in detail the seminal HCI concepts we leveraged to build our GUI fault model. Based on these concepts, the proposed GUI fault model is then detailed. Subsequently, the benefits of our proposal are highlighted through: an empirical analysis of existing GUI bug reports; the manual creation of GUI mutants on an existing system; and an evaluation of two GUI testing tools to detect such mutants. This paper ends with related work and the conclusion presenting GUI testing challenges.

II. Seminal HCI Concepts

Identifying GUI faults requires an examination in detail of the major HCI concepts. In this section we detail these concepts to highlight and explain in Section III the resulting GUI faults.

[1]Also called *command* [1], [2] or *event* [3].

[2]WIMP stands for *Windows, Icons, Menus, and Pointing device.*

[3]These interactions are more complex from a software engineering point of view. From a human point of view they should be more natural, *i.e.* more close to how people interact with objects in the real life.

Before introducing these seminal HCI concepts, we recall the basic elements that compose GUIs. Users act on an interactive system by performing a *user interaction* on a GUI. A user interaction produces as output an *action* that modifies the state of the system. For example, the user interaction that consists of pressing the button "*Delete*" of a drawing editor produces an action that deletes the selected shapes from the drawing. A user interaction is composed of a sequence of events (mouse move, *etc.*) produced by input devices (mouse, *etc.*) handled by users. One interaction may involve several input devices, which is then called a multi-modal interaction. For instance, pointing a position on a map and speaking to perform an action is a multi-modal interaction. The correct synchronization between the different input devices is a key concern and is called multi-modal fusion. A GUI is composed of graphical components, called widgets, laid out following a specific order. The graphical elements displayed by a widget are either purely aesthetics (fonts, *etc.*) or presentations of data. The state of a widget can evolve in time having effects on its graphical representation (*e.g.* visibility, position, value, data content).

Direct manipulation is one of the seminal HCI concepts [12], [13]. It aims at minimizing the mental effort required to use systems. To do so, direct manipulation promotes several rules to respect while developing GUIs. One of these rules stipulates that users have to feel engaged with control of objects of interest, not with GUIs or systems themselves. An example of direct manipulation is the drawing area of drawing editors. Such a drawing area represents shapes as 2D/3D graphical objects as most of the people define the concept of shapes. Users can handle these shapes by interacting *directly* within the drawing area to move or scale shapes using advanced interactions such as bi-manual interactions. Direct manipulation is in opposition to the use of standard widgets that bring indirection between users and their objects of interest. For instance, scaling a shape using a bi-manual interaction on its graphical representation is more direct than using a text field. So, developing direct manipulation GUIs usually implies the development of *ad hoc* widgets, such as the drawing area. These *ad hoc* widgets are usually more complex than standard ones since they rely on: advanced interactions (*e.g.* bi-manual, speech+pointing interactions); a dedicated data representation (*e.g.* shapes painted in the drawing area). Testing such heterogeneous and *ad hoc* widgets is thus a major challenge.

This contrast between GUIs composed of standard widgets only and GUIs that contain advanced widgets is reified, respectively, under the terms WIMP and post-WIMP. Van Dam proposed that a post-WIMP GUI is one "*containing at least one interaction technique not dependent on classical 2D widgets such as menus and icons*" [10].

Another seminal HCI concept is *feedback* [13], [14], [2], [9]. Feedback is provided to users while they interact with GUIs. It allows users to evaluate continuously the outcome of their interactions with the system. Feedback is computed and provided by the system through the user interface and can take many forms. A first simple example is when users move the cursor over a button. To notify that the cursor is correctly positioned to interact with the button this changes its shape. A more sophisticated example is the selection process of most of drawing editors that can be done using a Drag-And-Drop (DnD) interaction. While the DnD is performed on the drawing area, a temporary rectangle is painted to notify users about current selection area.

Another HCI concept is the notion of *reversible actions* [12], [13], [9]. The goal of reversible actions is to reduce user anxiety by about making mistakes [12]. In WIMP GUIs, reverting actions is reified under the undo/redo features usually performed using buttons or shortcuts that revert the latest executed actions. In post-WIMP GUIs, recent outcomes promote the ability to cancel actions in progress [15].

All these HCI concepts introduced in this section are interactive features that must be tested. However, we demonstrate in this paper that current GUI fault models and GUI testing tools do not cover all these features. In the next section, the GUI faults stemming from WIMP and post-WIMP GUIs are detailed.

III. Fault Model

In this section we present an exhaustive GUI fault model.

Bochmann *et al.* [11] define a fault model as:

Definition 1 (Fault Model): A fault model describes a set of faults responsible for a failure possibly at a higher level of abstraction.

To recall what a fault is:

Definition 2 (Fault): Faults are textual (or graphical) differences between an incorrect and a correct behavior description [16].

Based on these definitions, we propose the following definitions of a GUI fault and failure:

Definition 3 (GUI Fault): GUI faults are differences between an incorrect and a correct behavior description of a GUI.

Definition 4 (GUI Error): A GUI error is an activation of a GUI fault that leads to an unexpected GUI state.

Definition 5 (GUI Failure): A GUI failure is a manifestation of an unexpected GUI state provoked by a GUI fault.

A GUI fault can be introduced at different levels of a GUI software (*e.g.* GUI code, GUI models). An illustration of a GUI fault is: a correct line of GUI code *vs* an incorrect line of GUI code. For example, a GUI fault can be activated when an *unexpected entry*, such as a wrong value into an input widget, is not *handled correctly* by its GUI code. So, an unexpected GUI state is manifested (*e.g.* a crash as a GUI failure) when a user clicks on a button after typing this entry.

To build the proposed fault GUI model we first analyzed the state-of-the-art of HCI concepts (see Section II). We then analyzed real GUI bug reports (different than those used in Section IV) to assess and to precise the fault model. We performed a round trip process between the analysis of HCI concepts and GUI bug reports until obtain a stable fault model.

The description of our fault model is divided into two groups: The user interface faults and the user interaction faults. The user interface faults refer to faults affecting the structure and the behavior of graphical components of GUIs. The user interaction faults refer to faults affecting the interaction process when a user interacts with a GUI.

TABLE I. User Interface Faults

Fault categories	ID	Faults	Possible failures
GUI Structure and Aesthetics	GSA1	Incorrect layout of widgets (*e.g.* alignment, dimension, orientation, depth)	The positions of 2 widgets are inverted. A text is not fully visible since the size of text field is too small. Rulers do not appear on the top of a drawing editor. The vertical lines for visualizing the precise position of shapes in the drawing editor are not displayed.
	GSA2	Incorrect state of widgets (*e.g.* visible, activated, selected, focused, modal, editable, expandable)	Not possible to click on a button since it is not activated. A window is not visible so that its widgets cannot be used. Not possible to draw in the drawing area of a drawing editor since it is not activated.
	GSA3	Incorrect appearance of widgets (*e.g.* font, color, icon, label)	The icon of a button is not visible. In a GUI of a power plant, the color reflecting the critical status of a pump is green instead of red.
Data Presentation	DT1	Incorrect data rendering (*e.g.* scaling factors, rotating, converting)	The size of a text is not scaled properly. In a drawing editor, a dotted line is painted as a dashed one. A rectangle is painted as an ellipse.
	DT2	Incorrect data properties (*e.g.* selectable, focused)	A web address in a text is not displayed as hyperlink.
	DT3	Incorrect data type or format (*e.g.* degree *vs* radian, float *vs* double)	The date is displayed with five digits (*e.g.* dd/mm/y) instead of 6 digits (*e.g.* dd/mm/yy). A text field displays an angle in radian instead of in degree.

A. *User Interface Faults*

GUIs are composed of widgets that can act as mediators to interact indirectly (*e.g.* buttons on WIMP GUIs) or directly (direct manipulation principle in post-WIMP GUIs) with objects of the data model. In this section, we categorize the user interface faults, *i.e.* faults related to the structure, the behavior, and the appearance of GUIs. We further break down user interface into two categories: the *GUI structure and aesthetics*, and the *data presentation* fault, as introduced below. Table I presents an overview of these faults and their potential failures.

1) GUI Structure and Aesthetics Fault: This fault category corresponds to unexpected GUI designs. Since GUIs are composed of widgets laid out following a given order, the first fault is the *incorrect layout of widgets* (GSA1). Possible failures corresponding to this fault occur when GUI widgets follow an unexpected layout (*e.g.* wrong size or position). The next fault concerns the *incorrect state of widgets* (GSA2). Widgets' behavior is dynamic and can be in different states such as visible, enabled, or selected. This fault occurs when the current state of a widget differs from the expected one. For example, a widget is unexpectedly visible. The following fault treats the *unexpected appearance of widgets* (GSA3). That concerns aesthetic aspects of widgets not bound to the data model, such as look-and-feels, fonts, icons, or misspellings.

2) Data presentation: In many cases, widgets aim at editing and visualizing data of the data model. For example with WIMP GUIs, text fields or lists can display simple data to be edited by users. Post-WIMP GUIs share this same principle with the difference that the data representation is usually *ad hoc* and more complex. For example, the drawing area of a drawing editor paints shapes of the data model. Such a drawing area has been developed for the specific case of this editor. That permits to represent graphically in a single widget complex data (*e.g.* shapes). In other cases, widgets aim at monitoring data only. This is notably the case for some GUIs in control commands of power plants where data are not edited but monitored by users. The definition of data representations is complex and error-prone. It thus requires adequate *data presentation* faults.

The first fault of this category is the *incorrect data rendering* (DT1). DT1 is provoked when data is converted or scaled wrongly. Possible failures for this fault are manifested by unexpected data appearance (*e.g.* wrong color, texture, opacity, shadow) or data layout (*e.g.* wrong position, geometry). The second fault concerns incorrect data properties (DT2). Properties define specific visualization of data such as selectable or focused. A possible failure is a web address that is not displayed as a hyperlink. The last fault (DT3) occurs when an incorrect data type or format is displayed. For instance, an angle value is displayed in radian instead of in degree.

B. *User Interaction Faults*

In this section, we introduce the faults that treat user interactions. The proposed faults are based on the characteristics of WIMP and post-WIMP GUIs detailed in the previous section. For each fault we separated our analysis into two parts. One dedicated to WIMP interactions and another one to post-WIMP interactions. WIMP interactions refer to interactions performed on WIMP widgets. They are simple and composed of few events (click[4], key pressed, *etc.*). Post-WIMP interactions refer to interactions performed on post-WIMP widgets. Such interactions are more complex since they can be multimodal, *i.e.*involve multiple input devices (gesture, gyroscope, multi-touch screen); be concurrent (*e.g.* in bi-manual interactions the two hands evolve in parallel); be composed of numerous events (*e.g.* multimodal interactions may be composed of sequences of pressure, move, and voice events). Such interactions can be modeled as finite-state machines [9], [17], [18]. Subsequently the direct manipulation principles, other particularities of post-WIMP interactions are that they aim at: being as natural as

[4]A click is one interaction composed of the event *mouse pressed* followed by the event *mouse released*. Its simple behavior has leaded to consider a click as an event itself.

TABLE II. USER INTERACTION FAULTS

Fault categories	ID	Faults	Possible failures
Interaction Behavior	IB1	Incorrect behavior of a user interaction	A bi-manual interaction developed for a specific purpose does not work properly. The synchronization between the voice and the gesture does not work properly in a voice+gesture interaction.
Action	ACT1	Incorrect action results	Translating a shape to a position (x,y) translates it to the position $(-x,-y)$. Setting the zoom level at 150%, sets it at 50%.
	ACT2	No action executed	Clicking on a button has no effect. Executing a DnD on a drawing area to draw a rectangle has no effect.
	ACT1	Incorrect action executed	Clicking on the button *Save* shows the dialogue box used for loading. Scaling a shape results in its rotation. Performing a DnD to translate shapes results in their selection.
Reversibility	RVSB1	Incorrect results of undo or redo operations	Clicking on the button *redo* does not re-apply the latest undone action as expected. Pressing the keys *ctrl+z* does not revert the latest executed action as expected.
	RVSB2	Reverting the current interaction in progress works incorrectly	Pressing the key "*Escape*" during a DnD does not abort this last. Saying the word "*Stop*" does not stop the interaction in progress.
	RVSB3	Reverting the current action in progress works incorrectly	Clicking on the button "*Cancel*" to stop the loading of the file previously selected does not work properly.
Feedback	FDBK1	Feedback provided by widgets to reflect the current state of an action in progress works incorrectly.	The progress bar that shows the loading progress of a file works incorrectly.
	FDBK2	The temporary feedback provided all along the execution of long interactions is incorrect.	Given a drawing editor, drawing a rectangle using a DnD interaction does not show the created rectangle during the DnD as expected.

possible; providing users with the feeling of handling data directly (*e.g.* shapes in drawing editors). Table II summarizes the *user interaction* faults and some of their potential failures for both WIMP and post-WIMP interactions. These faults are detailed as follows.

1) Interaction Behavior: Developing post-WIMP interactions is complex and error-prone. Indeed, as explained in the section on GUIs' characteristics, it may involve many sequences of events or require the fusion of several modalities such as voice and gesture. So, the first fault (IB1) occurs when the behavior of the developed interactions does not work properly. This fault mainly concerns post-WIMP widgets since WIMP widgets embed simple and hard-coded interactions. For instance, an event such as *pressure* can be missing in a bi-manual interaction. Another example is the incorrect synchronization between the voice and the gesture in a voice+gesture interaction.

2) Action: This category of faults groups faults that concern actions produced while interacting with the system. The first fault (ACT1) focuses on the incorrect results of actions. In this case the expected action is executed but its results are not correct. For instance with a drawing editor, a failure can be the translation of one shape to the given position $(-x,-y)$ while the position (x,y) was expected. The root cause of this failure can be located in the action itself or in its settings. For instance, a first root cause of the previous failure can be the incorrect coding of the translation operation. A second root cause can be located in the settings of the translation action.

The second fault (ACT2) concerns the absence of action when interacting with the system. For instance, this fault can occur when an interaction, such as a keyboard shortcut, is not correctly bound to its widget.

The third fault (ACT3) consists of the execution of wrong actions. The root cause of this fault can be that the wrong action is bound to a widget at a given instant. For instance: clicking on the button *Save* shows the dialogue box used for loading; doing a DnD interaction on a drawing area selects shapes instead of translating them.

3) Reversibility: This fault category groups three faults. The first fault (RVSB1) concerns the incorrect behavior of the undo/redo operations. Undo and redo operations usually rely on WIMP widgets such as buttons and key shortcuts. These operations revert or re-execute actions *already terminated* and stored by the system. A possible failure is the incorrect reversion of the latest executed action when the key shortcut *ctrl+z* is used.

Contrary to WIMP interactions, that are mainly one-shot, many interactions last some time such as the DnD interaction. In such a case, users may be able to stop an interaction in progress. The second fault (RVSB2) thus consists of the incorrect interruption of the current interaction in progress. For instance, pressing the key "*Escape*" during a DnD does not stop this last. This fault could have been classified as an interaction behavior fault. We decided to consider it as a reversibility fault since it concerns the ability to revert an ongoing interaction.

Once launched, actions may take time to be executed entirely. In this case such actions can be interrupted. The third fault (RVSB3) concerns the incorrect interruption of an action in progress. A possible failure concerns the file loading operation: clicking on the button "*Cancel*" to stop the loading of a file does not work properly.

4) Feedback: Widgets are designed to provide immediate and continuous feedback to users while they interact with them. For instance, progress bars showing the loading progress of a file is a kind of feedback provided to users. The first fault of this category (FDBK1) concerns the incorrect feedback provided by widgets to reflect the current state of an action in progress. This fault focuses on actions that last in time and *which progress should be monitored* by users.

The second fault (FDBK2) focuses on potentially long interactions (*i.e.* interactions taking a certain amount of time to be completed) which progress should be discernible by users. For instance with a drawing editor, when drawing a shape on the drawing area, the shape in creation should be visible so that the user knows the progression of her work. So, a possible failure is drawing a rectangle using a DnD interaction, that works correctly, does not show the created rectangle during the DnD as expected.

C. Discussion

The definition and the use of a fault model raise several questions we discuss about in this sub-section.

What are the benefits of the proposed GUI fault model? The benefits of our GUI fault model are twofold. First, a fault model is an exhaustive classification of faults for a specific concern [11]. Providing a GUI fault model permits GUI developers and testers to have a precise idea of the different faults they must consider. As an illustration, Section IV describes an empirical analysis we conducted to classify and discuss about GUI failures of open-source GUIs. Second, our GUI fault model allows developers of GUI testing tools to evaluate the efficiency of their tool in terms of bug detection power w.r.t. a GUI specific fault model. As detailed in Section VI, we created mutants of an existing GUI. Each mutant contains one GUI failure that corresponds to one GUI fault of our fault model. Developers of GUI testing tools can run their tools against these mutants for benchmarking purposes.

Should usability have been a GUI fault? Answering this question requires the definition of a fault to be re-explained: a fault is a difference between the observed behavior description and the expected one. Usability issues consist of reporting that the current observed behavior of a specific part of a GUI lacks at being somehow usable. That does not mean the observed behavior differs from the behavior expected by test oracles. Instead, it usually means that the expected behavior has not been defined correctly regarding some usability criteria. That is why we do not consider usability as a GUI fault. This reasoning can be extended to other concerns such as performance.

How to classify GUI failures into a fault model? A GUI failure is a perceivable manifestation of a GUI error. Classifying GUI failures thus requires to have identified the root cause (*i.e.* GUI error) of the failure. So, classifying GUI failures can be done by experts of the GUI under test. These experts need sufficient information, such as patches, logs, or stack traces, to identify if the root cause of a failure is a GUI error to then classify it. For example, given a failure manifested in the GUI and caused by a precondition violation. In this case, such a failure is not classified into the GUI fault model. Similarly, classifying correctly a GUI failure also requires to qualify the involved widgets (*e.g.* standard or *ad hoc*) as well as the interaction (*e.g.* mono-event or multiple-event interaction).

How to classify failures stemming from other failures? For instance, the incorrect results of the execution of an action (action fault) let a widget not visible as expected (GUI structure fault). In such cases, only the first failure must be considered since it puts the GUI in an unexpected and possibly unstable state. Besides, the appearance of a GUI error depends on the previous actions and interactions successfully executed. Typical examples are the undo and redo actions. A redo action can be executed only if an action has been previously performed. Furthermore, the success of a redo action may depend on the previous executed actions. We considered this point during the creation of mutants (as detailed in Section VI) to provide failures that appear both with and without previous actions.

IV. Relevance of the Fault Model: an Empirical Analysis

In this section the proposed GUI fault model is evaluated. Our evaluation has been conducted by an empirical analysis to assess the relevance of the model w.r.t. faults currently observed in existing GUIs. The goal is to state whether our GUI fault model is relevant against failures found in real GUIs.

A. Introduction

To assess the proposed fault model, we analyzed bug reports of 5 popular open-source software systems: Sweet Home 3D, File-roller, JabRef, Inkscape, and Firefox Android. These systems implement various kinds of widgets, interactions, and encompass different platforms (desktop and mobile). Their GUIs cover the main following features: *indirect and direct* manipulation; *several input devices* (*e.g.* mouse, keyboard, touch); *ad hoc widgets* such as canvas; *discrete data manipulation* (*e.g.* vector-based graphics); and *undo/redo* actions.

B. Experimental Protocol

Bug reports have been analyzed manually from the researcher/tester perspective by looking only at data available in the failures report (*i.e.* black box analysis). To focus on detailed and commented bug reports that concern GUI failures, the selection has been driven by the following rules. Only closed, fixed, and in progress bug reports were selected. The following search string has been also used to reduce the resulting sample: *interface OR "user interface" OR “graphical user interface” OR "graphical interface" OR GUI OR UI OR layout OR design OR graphic OR interaction OR “user interaction” OR interact OR action OR feedback OR revert OR reversible OR undo OR redo OR abort OR stop OR cancel*. Each report has been then

TABLE III. DISTRIBUTION OF ANALYZED FAILURES PER SOFTWARE

Software	Analyzed failures	User interface failures	User interaction failures	Repositories link
Sweet Home 3D	33	55%	45%	http://sourceforge.net/p/sweethome3d/bugs/
File-roller	32	28%	72%	https://bugzilla.gnome.org/query.cgi
JabRef	84	42%	58%	http://sourceforge.net/p/jabref/bugs/
Inkscape	82	28%	72%	https://bugs.launchpad.net/inkscape/
Firefox Android	48	60%	40%	https://bugzilla.mozilla.org/

manually analyzed to state whether it is a GUI failure. Also, selected bug reports have to provide explanations about the root cause of the failure such as a patch or comments. This step is crucial to be able to categorize the failures using our GUI fault model considering their root cause. We also discarded failures identified as non-reproducible, duplicated, usability, or user misunderstanding. From this selection we kept 279 bug reports (in total for the five systems) describing one GUI failure each. The following sub-sections discuss about these failures and the classification process.

C. Classification and Analysis

All the 279 failures have been successfully classified into our fault model. Fig. 1 gives an overview of the selected bug reports classified using our proposed fault model. These failures were classified into the *Action* (119 failures, 43%), *GUI Structure and Aesthetics* (75 failures, 27%), *Data Presentation* (39 failures, 14%), *Reversibility* (31 failures, 11%), *Interaction behavior* (12 failures, 4%), and *Feedback* (3 failures, 1%) fault categories. Most of the failures classified into *GUI Structure and Aesthetics* concern the *incorrect layout of widgets* (51%). Likewise, most of the failures in the *Action* category refer to *incorrect action results* (75%).

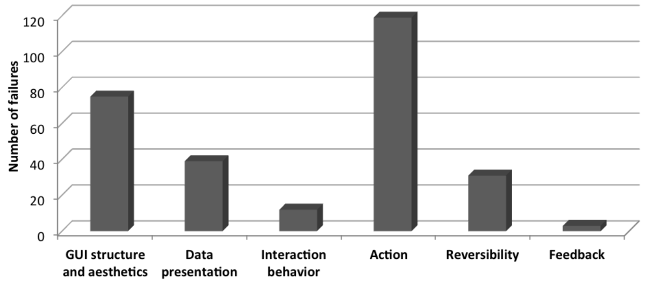

Fig. 1. Classification of the 279 bug reports using the GUI fault model

Table III shows the distribution of the 279 analyzed GUI failures per software and category (user interface or user interaction). These results point out that the systems *Sweet Home 3D* and *Firefox Android* seem to be more affected by user interface failures. Most of these failures concern the *GUI structure and aesthetics* fault. That can be explained by the complex and *ad hoc* GUI structure of these systems. *File Roller* and *JabRef* GUIs include widgets with coarse-grained properties (*i.e.* simple input value such as number or text). Most of their failures concern WIMP interactions classified into the *action* category. In contrast, *Inkscape* presented more failures classified as post-WIMP. Indeed, Inkscape, a vector graphics software, mainly relies on its drawing area that provides users with different post-WIMP interactions. These failures have been categorized mainly into *interaction behavior*, *action*, and *reversibility*.

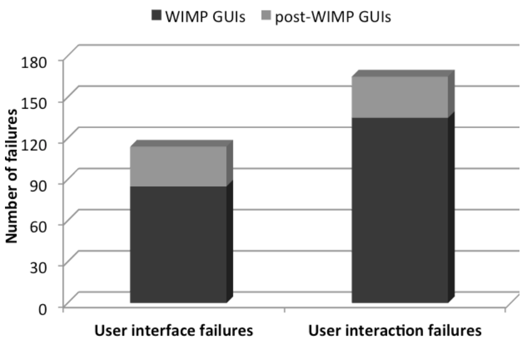

Fig. 2. Manifestation of failures in the user interface and user interaction levels

As depicted by Fig. 2, 41% of these 279 GUI failures are originated by faults classified into the user interface category and 59% into the user interaction category. Most of user interaction failures have been classified into the *incorrect action results* (54%). This plot also highlights that only 25% of the analyzed user interface failures and 18% of the user interaction ones have been classified as post-WIMP. We comment these results in the following sub-section.

D. Discussion

The empirical results must be balanced with the fact that user interactions are less tangible than user interfaces. So, users may report more GUI failures when they can perceive failures graphically (an issue in the layout of a GUI or in the result of an action visible through a GUI). Users, however, may have difficulties to detect a failure in an interaction itself while interacting with the GUI. That may explain the low number of failures (4%) classified into *Interaction Behavior*. Another explanation may be the primary use of WIMP widgets, relying on simple interactions.

In our analysis, many failures that could be related to *Feedback* were discarded since they concerned enhancements or usability issues, which are out of the scope of a GUI fault model as discussed previously. For instance, GUI failures that concern the lack of haptic feedback in Firefox Android were discarded. So, few faults (1%) were classified into this category. Another explanation may be the difficulty for users to identify feedback issues as real failures that should be reported.

We observed that some reported GUI failures are false positives regarding the *fault localization*: if the report does not have enough information about the root cause of a failure (*e.g.* patch or exception log), a GUI failure can be classified in a wrong fault category. For example, when moving a shape using a DnD does not move it. At a first glance, the root cause of this failure can be associated to an incorrect behavior of the DnD. So, this failure can be categorized into the interaction behavior. However, after analyzing the root cause this failure refers to an action failure since the DnD works properly, but no action is linked to this interaction.

Likewise, the failures related to *Reversibility* and *Feedback* were easily identified through the steps to reproduce them. For example in JabRef, "*pressing the button "Undo" will clear all the text in the field, but then pressing the button "Redo" will not recover the text*". Furthermore, some systems do not revert interactions step by step but entirely. This can imply a failure from a user's point view, but sometimes it is considered as an invalid failure (*e.g.* requirements vs. usability issues) by developers. In *JabRef*, the undo/redo actions did not revert discrete operations. For example, pressing the button "Undo" clears all texts typed into different text fields instead of clearing only one field each time the button "undo" is pressed.

Another important point concerns the WIMP vs. post-WIMP GUIs faults. We classified more failures involving WIMP than post-WIMP widgets. A possible explanation is that, despite the increasing interactivity of GUIs, the analyzed GUIs still rely more on WIMP widgets and interactions. Moreover, users now master the behavior of WIMP widgets so that they can easily identify when they provoke failures. It may not be the case with *ad hoc* and post-WIMP widgets.

V. Are GUI Testing Tools Able to Detect Classified Failures? An Empirical Study

This section provides an empirical study of two GUI testing tools: GUITAR [19] and Jubula[5]. To demonstrate the current limitations of GUI testing tools in testing real GUIs, we applied those tools to detect the failures previously classified into our GUI fault model.

A. GUITAR and Jubula

GUITAR is one of the most widespread academic GUI testing tools. It extracts the GUI structure by reverse engineering. This structure is transformed into a GUI Event Flow Graph (EFG), where each node represents a widget event. Based on this EFG, test cases are generated and executed automatically over the SUT. We used the plugin for Java Swing (*i.e.* JFC GUITAR version 1.1.1)[6]. In GUITAR, each test case is composed by a sequence of widget events. The generation of test cases can be parameterized with the size of that sequence (*i.e.* test case length).

Jubula is a semi-automated GUI testing tool that leverages pre-defined libraries to create test cases. These libraries contain modules that can be reused to generate manually test sequences. The modules encompass actions (*e.g.* check, select) and interactions (*e.g.* click, drag and drop) over different GUI toolkits (*e.g.* swing, SWT, RCP, mobile). We have reused the library dedicated to Java Swing (Jubula version 7.2) to write the test cases presented in the next experiments. This library contains actions to test only standard widgets such as dragging a column/row of a table by passing an index. To test *ad hoc* widgets (*e.g.* canvas), we made a workaround by mapping actions directly to these widgets. For example, to draw a shape on canvas we need to specify the exact position (*e.g.* drag and drop coordinates) where the interaction should be executed.

[5]http://www.eclipse.org/jubula

[6]http://sourceforge.net/apps/mediawiki/guitar/

B. Experiment

We selected JabRef[7], a software to manage bibliographic references. JabRef is written in Java which allows us to apply both GUITAR and Jubula. For each fault described in our GUI fault model, we selected one reported failure. To reproduce each failure, we downloaded the corresponding faulty version of JabRef. We used the exact test sequence (*i.e.* number of actions) to reproduce a failure. In GUITAR, all test cases were generated automatically over a faulty version. In Jubula, each test case was created manually to detect one failure. All test cases were written by one of the authors of this paper who has expertise in JabRef. Also, their test sequences are extracted by analyzing failure reports (*e.g.* steps to reproduce a failure) and reusing Jubula's libraries. Then, GUITAR and Jubula run all their test cases automatically for checking whether the selected failure is found.

C. Results and Discussion

Table V summarizes the detection of the JabRef GUI failures by GUITAR and Jubula. These failures cover 11 out of the 15 faults described in our fault model. The remaining four faults were not covered for two reasons: 1) no failure was classified for that fault; or 2) a failure was classified, but we could not reproduce it - only occurred in a specific environment (*e.g.* Operating System) or given a certain input (*e.g.* a particular database in JabRef).

The reported failures in JabRef are mostly related to WIMP widgets, so we would expect GUITAR and Jubula to detect them, but it was not the case. For instance, failure #1 reports an incorrect display of buttons' label; its root cause is the incorrect size of a widget positioned to the left of them. Thus, this failure does not affect the values of internal properties (*e.g.* text, event handlers) of those buttons. In GUITAR, checking the properties of that widget did not reveal this failure since the expected and actual values of its size property (*e.g.* width) remained the same. In Jubula, the concerned widget cannot be mapped to test cases execution and thus cannot be tested.

Failures #2 and #3 refer to an incorrect menu path and a misspelling, respectively. Both failures were detected by Jubula. However, these failures were not found by GUITAR. Indeed, GUITAR does reverse engineering of an existing GUI to produce tests. If this GUI is faulty, GUITAR will produce tests that will consider these failures as the correct behavior.

Failures #8 and #13 that lead to a crash of the GUI were found by both GUITAR and Jubula. However, failures #4, #6, #10, and #11 that affect the data model were not detected by GUITAR for two reasons. First, GUITAR does not test the table entries in JabRef since they represent the data model. To do this, we need to extend GUITAR to interact with them. Second, the test cases successfully passed, but a failure has been revealed. That means, the events are fired properly (*e.g.* no exception) and GUI properties are the "expected" ones. For example, a text property of a status bar contains the value: *"Redo: change field"*, when this action was actually not redone. Similarly, failure #10 was not detected by Jubula. This failure reports an unexpected auto-completion when the action "save" is triggered by shortcuts. We reproduced this failure manually

[7]http://jabref.sourceforge.net/

TABLE IV. MUTANTS PLANTED ACCORDING TO FAULTS IN THE GUI FAULT MODEL

ID	GSA1	GSA2	GSA3	DT1	DT2	DT3	IB1	ACT1	ACT2	ACT3	RVSB1	RVSB2	RVSB3	FDBK1	FDBK2
#Mutants	3	7	4	2	1	1	1	8	16	5	9	2	-	3	3
#Length	0..2	0..4	0..4	4	2	4	4	1..7	1..8	4..5	2..8	2	-	2..5	3..4

TABLE V. JABREF FAILURES DETECTED BY GUITAR AND JUBULA

ID fault	**ID failure**	**Bug repository link**	**GUITAR**	**Jubula**
GSA1	#1	http://sourceforge.net/p/jabref/bugs/160/	✗	✗
GSA2	#2	http://sourceforge.net/p/jabref/bugs/514/	✗	✓
GSA3	#3	http://sourceforge.net/p/jabref/bugs/166/	✗	✓
DT1	#4	http://sourceforge.net/p/jabref/bugs/716/	✗	✓
DT2	#5	-		
DT3	#6	http://sourceforge.net/p/jabref/bugs/575/	✗	✓
IB1	#7	-		
ACT1	#8	http://sourceforge.net/p/jabref/bugs/495/	✓	✓
ACT2	#9	http://sourceforge.net/p/jabref/bugs/536/	✓	✓
ACT3	#10	http://sourceforge.net/p/jabref/bugs/809/	✗	✗
RVSB1	#11	http://sourceforge.net/p/jabref/bugs/560/	✗	✓
RVSB2	#12	-		
RVSB3	#13	http://sourceforge.net/p/jabref/bugs/458/	✓	✓
FDBK1	#14	http://sourceforge.net/p/jabref/bugs/52/	✗	✓
FDBK2	#15	-		

but the test case was successfully replayed by Jubula. The input text via keyboard was typed and saved automatically without any interference of the auto-completion feature.

Another point is the accuracy of test cases generated manually in Jubula. Detecting failure #6 depends on how the test case is written. For example, adding a field that contains LaTeX commands (*e.g.* 100\%), and then checking its output in a preview window should not contain any command (*e.g.* 100%). So, we can write a test case to test the outputs in the preview window only looking for commands (*e.g. SelectPattern[%, equals] in ComponentText[preview]*). Or, write a test case to check whether an entire text matches to the expected one (*e.g. CheckText[100%, equals] in ComponentText[preview]*). However, the last test case will fail since a text from preview window in JabRef is shown internally as HTML and, in Jubula, the action's parameters cannot be specified in that format.

Our experiment does not aim at comparing both tools since GUITAR is a fully automated tool contrary to Jubula. However, the results of this study highlight the current limitations of GUI testing tools. GUITAR and Jubula currently mainly work for detecting failures that affect properties of standard widgets. Moreover, GUITAR does GUI regression testing: it considers a given GUI as the reference one from which tests will be produced. If this GUI is faulty, GUITAR will produce tests that will consider these failures as the correct behavior. A possible solution to overcome this issue is to base the test process on the specifications (requirements, *etc.*) of the GUI.

VI. FORGING FAULTY GUIS FOR BENCHMARKING

In this section, we evaluate the usefulness of our fault model by applying it on a highly interactive open-source software system. We created mutants of this system corresponding to the different faults of the model. The main goal of these mutants is to provide GUI testers with benchmark tools to evaluate the ability of GUI testing tools to detect GUI failures. As an illustration of the practical use of these mutants, we executed two GUI testing tools against the mutants of the system. Thanks to that we caught a glimpse of their ability to cover our proposed fault model. The goal of this experiment is to answer the research question: *what are the benefits of this fault model for GUI testing?*

A. Mutants Generation

As highlighted by Zhu *et al.*, *"software testing is often aimed at detecting faults in software. A way to measure how well this objective has been achieved is to plant some artificial faults into the program and check if they are detected by the test. A program with a planted fault is called a mutant of the original program"* [20]. Following this principle, we planted 65 faults in a highly interactive open-source software system, namely Latexdraw[8], using our proposed fault model. Latexdraw has been selected because of the following points: 1) it is a highly interactive system written in Java and Scala (dedicated to the creation of drawings for LaTeX); 2) its GUI mixes both standard and *ad hoc* widgets; 3) it is released under an open-source license (GPL2) so that it can be freely used by the testing community.

We created 65 mutants corresponding to the different faults of our proposed fault model. All these mutants and the original version are freely available[9]. Each mutant is documented to detail its planted fault and the oracle permitting to find it[9]. Multiple mutants have been created from each fault by: using WIMP (22 mutants) or post-WIMP (43 mutants) widgets to kill the mutants; varying the test case length (*i.e.* the number of actions required to provoke the failure). Each action (*e.g.* select a shape) requires a minimal number of events (*e.g.* in LaTeXDraw a DnD requires at least three events: press/move/release) to be executed. Table IV summarizes the number of forged mutants and the minimal and maximal test case length for each fault. For instance, a length 0..2 means there exists at least one mutant requiring a minimum of 0 action or a maximum of two actions). However, the fault RVSB3 is currently not covered by the Latexdraw mutants. Similarly, some planted mutants only rely on post-WIMP interactions or widgets (*e.g.* IB1, DT1).

B. How GUI testing tools kill our GUI mutants: a first experiment

We applied the GUI testing tools GUITAR and Jubula on the mutants to evaluate their ability to kill them. Our goal is not to provide benchmarks against these tools but rather highlight the current challenges in testing interactive systems not considered yet (*e.g.* post-WIMP interactions). GUITAR test cases have been generated automatically while Jubula ones have been written manually.

Considering the mutants planted at the user interface level, Jubula and GUITAR tests killed the mutants that involve checking standard widget properties, such as layout (*e.g.* width,

[8]http://sourceforge.net/projects/latexdraw/

[9]https://github.com/arnobl/latexdraw-mutants

height) and state (*e.g.* enable, selection, focusable). Also, it is possible to test simple data (*e.g.* string values on text fields) on those widgets. However, most of the mutants that concern the *ad hoc* widgets were alive. Notably, when test cases involve testing complex data from the data model. For example, it is not possible to compare the results of the actual shape on canvas against the expected one. Even if some shape properties (*e.g.* rotation angle) are presented on standard widgets (*e.g.* spinner), GUITAR and Jubula cannot state whether the current values in these widgets match the expected shape rotation on the canvas.

Likewise, our GUITAR and Jubula tests cannot kill most of the user interaction mutants that result on a wrong presentation of shapes. In particular, when we tested mutants planted into the *Reversibility* or *Feedback* categories. For example, testing undo/redo operations in Latexdraw should compare all states to manipulate a shape on canvas. Moreover, the tests verdict on Jubula passed even though interactions are defined incorrectly (*e.g.* mouse cursor does not follow a DnD) or actions cannot be executed (*e.g.* a button is disabled). In GUITAR, the generated test cases do not cover properly actions having dependencies. For example, the action "*Delete*" in Latexdraw requires first selecting a shape on canvas. However, no test sequence that contains "*Select Shape*" before "*Delete Shape*" was generated. Thus, some mutants could not be killed.

Table VI gives an overview of the number of mutants killed by GUITAR and Jubula. The results show that both tools are not able to kill all mutants because of the four following reasons: 1) *Testing Latexdraw with GUITAR and Jubula is limited to the test of the standard Swing widgets*. In Jubula, the test cases are only written according to libraries available in the Swing toolkit. In GUITAR, the basic package for Java Swing GUIs only covers standard widgets and mono-events (*e.g.* a click on a button). 2) *Configure or customize a GUI testing tool to test post-WIMP widgets is not a trivial task*. For example, each sequence of a test case in Jubula needs to be mapped for the corresponding GUI widget manually. Also, GUITAR needs to be extended to generate test cases for *ad hoc* widgets (*e.g.* canvas) as well their interactions (*e.g.* multi-modal interactions). 3) *Testing post-WIMP widgets requires a long test case sequence*. In Latexdraw, a sequence to test interactions over these widgets is composed of at least two actions. That sequence is longer when we have to detect failures into undo/redo operations. 4) *It is not possible to give a test verdict for complex data*. The oracle provided by the two GUI testing tools do not know the internal behavior of *ad hoc* widgets, their interaction features and data presentation. These results answer the research question by highlighting the benefits of our fault model for measuring the ability of GUI testing tools in finding GUI failures.

C. Threats to Validity

Regarding the conducted empirical studies, we identified the two following threats to validity. The first one concerns the scope of the proposed fault model since we evaluated it empirically on a small number (five) of interactive systems. To limit this threat, we selected interactive systems that cover different aspects of the HCI concepts we detailed in Section II. The second threat we identified concerns the subjectivity observed in bug reports to describe failures. To deal with this, we based the classification on the bug report artifacts (patches, logs, *etc.*) to identify the root cause of the reported failures.

TABLE VI. MUTANTS KILLED BY GUITAR AND JUBULA

	GUITAR		**JUBULA**	
ID	WIMP	post-WIMP	WIMP	post-WIMP
GSA1	2	0	2	0
GSA2	5	0	6	1
GSA3	3	0	3	0
DT1	-	0	-	0
DT2	-	0	-	0
DT3	-	0	-	1
IB1	-	0	-	0
ACT1	0	0	0	1
ACT2	3	0	3	0
ACT3	2	0	2	0
RVSB1	2	0	2	0
RVSB2	-	0	-	0
RVSB3	-	-	-	-
FDBK1	1	0	1	0
FDBK2	-	0	-	0

VII. Related Work

Existing fault classifications are presented in a higher level of abstraction considering mainly the components that are affected by faults. Most classifications leverage the software assets (*e.g.* specification, models, architecture, code) to define their faults. These faults have been described into fault models [11], [16] or defects taxonomies [21].

In an effort to cover GUIs, the Orthogonal Defect Classification (ODC) [21] is extended by IBM Research to include GUIs faults. These faults focus on the appearance of widgets, navigation between widgets, unexpected behavior of widgets events and input devices. In our fault model, we do not cover faults that concern the behavior of input devices (*i.e.* hardware fault). Although this taxonomy considers GUI faults, it does not separate the user interface and user interaction faults. Moreover, this extension does not consider faults caused by post-WIMP widgets and their advanced interactions as well faults into the *data presentation* category.

Li *et al.* categorize faults of industrial and open source projects using the ODC taxonomy [22]. The category *Interface* concerns several GUI defects. However, this single category covers several user interface defects related to specific widgets such as *window*, *title bar*, *menu*, or *tool bar*. Similarly, the interaction defects are limited to *mouse* and *keyboard*. Thus, it is not possible to identify the kind of faults classified into these categories since they are not detailed. For example, a fault classified into the *mouse* category can concern an interaction, an action, or an input device.

Brooks *et al.* [23] present a study that characterizes GUIs based on reported faults of three industrial systems. To classify all these faults (GUI and non-GUI faults), the authors adapted a defect taxonomy by including other categories such as GUI defects. This category encompasses both the user interface and user interaction faults. Also, Børretzen *et al.* [24] analyze faults reported by four projects by combining two defect taxonomies. Both works introduce a category that concerns the GUI faults but these faults are not described and thus no classification is presented. Strecker *et al.* [25] characterize faults that affect GUI test suites. However, these faults do not concern the GUI faults but any fault at the code level (*e.g.* class or method faults) that may affect the GUI.

In contrast, several research papers concern the fault effects by classifying GUI failures instead of GUI faults. In general, these works focus on specific GUIs (automotive GUIs [26]) or domains (mobile [27], safety-critical [28]). For example, Maji *et al.* characterize failures for mobile operating systems [27]. These failures are classified according to the fault localization. For example, a failure manifested in a *camera* is categorized in the *Camera segment*. Similarly, failures for other segments such as Web, Multimedia, or GUI are categorized. Also, Zaeem *et al.* [29] have conducted a bug study for Android applications to automate oracles. They identified 20 categories including some GUI issues such as Rotation (device's rotation), Gestures (zooming and out) and Widget. Although, these papers have investigated failures in a context that brings many advances in terms of interactive features, no classification or discussion about these kinds of failures is presented.

Mauser *et al.* propose a GUI failure classification for automotive systems [26]. This classification is based on the three categories: design, content, and behavior. In the *Design* category, the failures refer to GUI layouts (*e.g.* color, font, position). In the *Content* category, the failures are associated to data displayed such as text, animation, and symbols/icons. The failures in the *Behavior* category are caused by a wrong behavior of windows (*e.g.* wrong pop-up) or widgets (*e.g.* wrong focus). The authors focus on characterizing GUI failures based only on a small set of specific widgets designed for these kinds of GUIs. Furthermore, they do not consider user interaction failures.

VIII. Conclusion and Research Agenda

This paper proposes a GUI fault model for providing GUI testers with benchmark tools to evaluate the ability of GUI testing tools to detect GUI failures. This fault model has been empirically assessed by analyzing and classifying into it 279 GUI bug reports of different open-source GUIs. To demonstrate the benefits of the proposed fault model, mutants have then been developed from it on a Java open-source GUI. As an illustrative use case of these mutants, we executed two GUI testing tools on these mutants to evaluate their ability to detect them. This experiment shows that if current GUI testing tools have demonstrated their ability for finding several kinds of GUI errors, they also fail at detecting several GUI faults we identified. The underlying reasons are twofold. First, GUI failures may be related to the graphical rendering of GUIs. Testing a GUI rendering is a complex task since current testing techniques mainly rely on code analysis that can hardly capture graphical properties. Second, the current trend in GUI design is the shift from designing GUIs composed of standard widgets to designing GUIs relying on more complex interactions and *ad hoc* widgets [2], [8], [9]. New GUI testing techniques have thus to be proposed for fully testing, *as automated as possible*, GUI rendering and complex interactions using *ad hoc* widgets.

Acknowledgements

This work is partially supported by the French BGLE Project CONNEXION.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[2] M. Beaudouin-Lafon, "Instrumental interaction: an interaction model for designing post-WIMP user interfaces," in *Proc. of CHI'00*. ACM, 2000, pp. 446–453.

[3] A. M. Memon, "An event-flow model of GUI-based applications for testing," *STVR*, vol. 17, no. 3, pp. 137–157, 2007.

[4] M. Cohen, S. Huang, and A. Memon, "Autoinspec: Using missing test coverage to improve specifications in GUIs," in *Proc of ISSRE'12*, 2012, pp. 251–260.

[5] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee, and A. Memon, "Lightweight static analysis for GUI testing," in *Proc of ISSRE'12*, 2012.

[6] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *Proc. of ICST'12*. IEEE, 2012, pp. 81–90.

[7] D. H. Nguyen, P. Strooper, and J. G. Süß, "Automated functionality testing through GUIs," in *Proc. of ACSC '10*, 2010, pp. 153–162.

[8] M. Beaudouin-Lafon, "Designing interaction, not interfaces," in *Proc. of AVI'04*, 2004.

[9] A. Blouin and O. Beaudoux, "Improving modularity and usability of interactive systems with Malai," in *Proc. of EICS'10*, 2010, pp. 115–124.

[10] A. van Dam, "Post-WIMP user interfaces," *Commun. ACM*, vol. 40, no. 2, pp. 63–67, Feb. 1997.

[11] G. von Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo, "Fault models in testing." in *Protocol Test Systems*, 1991, pp. 17–30.

[12] B. Shneiderman, "Direct manipulation: a step beyond programming languages," *IEEE Computer*, vol. 16, no. 8, pp. 57–69, 1983.

[13] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct manipulation interfaces," *Hum.-Comput. Interact.*, vol. 1, no. 4, pp. 311–338, 1985.

[14] D. A. Norman, *The Design of Everyday Things*, reprint paperback ed. Basic Books, 2002.

[15] C. Appert, O. Chapuis, and E. Pietriga, "Dwell-and-spring: undo for direct manipulation," in *Proc. of CHI'12*. ACM, 2012, pp. 1957–1966.

[16] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar, "A generic fault model for quality assurance," in *Proc of MODELS'13*, 2013.

[17] A. Blouin, B. Morin, G. Nain, O. Beaudoux, P. Albers, and J.-M. Jézéquel, "Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation," in *Proc. of EICS'11*, 2011, pp. 85–94.

[18] C. Appert and M. Beaudouin-Lafon, "SwingStates: Adding state machines to Java and the Swing toolkit," *SW: Practice and Experience*, vol. 38, no. 11, pp. 1149–1182, 2008.

[19] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.

[21] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[22] N. Li, Z. Li, and X. Sun, "Classification of software defect detected by black-box testing: An empirical study," in *Proc. of WCSE'10*.

[23] P. Brooks, B. Robinson, and A. Memon, "An initial characterization of industrial graphical user interface systems," in *Proc. of ICST'09*.

[24] J. A. Børretzen and R. Conradi, "Results and experiences from an empirical study of fault reports in industrial projects," in *Proc. of PROFES'06*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 389–394.

[25] J. Strecker and A. Memon, "Relationships between test suites, faults, and fault detection in gui testing," in *Proc. of ICST'08*, 2008, pp. 12–21.

[26] D. Mauser, A. Klaus, R. Zhang, and L. Duan, "GUI failure analysis and classification for the development of in-vehicle infotainment," in *Proc. of VALID'12*, 2012, pp. 79–84.

[27] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *Proc. of ISSRE'10*, 2010, pp. 249–258.

[28] R. Lutz and I. mikulski, "Empirical analysis of safety-critical anomalies during operations," *IEEE Trans. Softw. Eng.*, pp. 172–180, 2004.

[29] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proc. of ICST'14*, 2014.