On Model-Based Testing Advanced GUIs

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

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

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

Fabien Coulon
Inria
Rennes, France
fabien.coulon@inria.fr

***Abstract*—Graphical User Interface (GUI) design is currently shifting from designing GUIs composed of standard widgets to designing GUIs relying on more natural interactions and *ad hoc* widgets. This shift is meant to support the advent of GUIs providing users with more adapted and natural interactions, and the support of new input devices such as multi-touch screens. Standard widgets (*e.g.* buttons) are more and more replaced by *ad hoc* ones (*e.g.* the drawing area of graphical editors), and interactions are shifting from mono-event (*e.g.* button pressures) to multi-event interactions (*e.g.* multi-touch and gesture-based interactions). As a consequence, the current GUI model-based testing approaches, which target event-based systems, show their limits when applied to test such new advanced GUIs. The work introduced in this paper establishes three contributions: a precise analysis of the reasons of these current limits; a proposition to tackle the identified limits by leveraging the Malai GUI specification language and by proposing the concept of interaction-action-flow graph; feedback from two use cases, an industrial project and an open-source application, where the proposed approach has been applied.**

Keywords*—GUI testing, model-based testing, human-computer interaction

I. Introduction

The constant increase of system interactivity requires software testing to closely consider the testing of graphical user interfaces (GUI). The standard GUI model-based testing process is depicted in Figure 1. The first step consists of obtaining models describing the structure and behavior of a GUI of the system under test (SUT) using a User Interface Description Language (UIDL). These models can be automatically extracted by reverse engineering from the SUT binaries [23]. Such a model of existing GUIs is effective for detecting crashes and regressions. However, in some cases, reverse engineering is not possible. For instance, the norms IEC 60964 and 61771, dedicated to the validation and design of nuclear power plants, require that: developed systems must conform to the legal requirements; models must be created from requirements in that purpose [17], [18]. In this case, GUI models are designed manually from the requirements, and the testing process targets mismatches between a system and its specifications. Once a model of the GUI is available, a test model is produced, to drive test generation. In GUI testing, test models are mainly event-flow graphs (EFG). EFGs contain all the possible sequences of user interactions that can be performed within a GUI. Test scripts are automatically generated by traversing the EFG according to a specific test adequacy criterion. Test scripts are executed on the SUT manually or automatically. Finally, GUI oracles yield test verdicts by comparing effective results of test scripts with the expected ones.

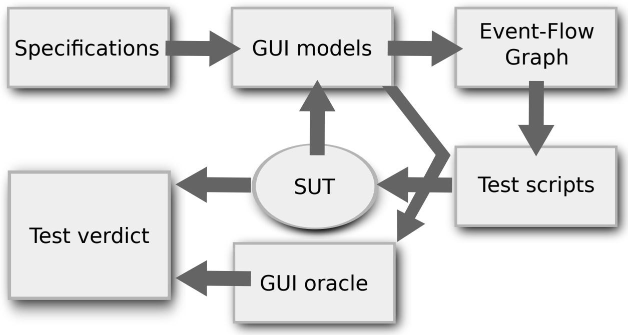


Fig. 1: Standard GUI model-based testing process

GUIs are composed of graphical objects called widgets, such as buttons. Users interact with these widgets (*e.g.* press a button) to product an action[1] that modifies the state of the system, the GUI, or the data model. For instance, pressing the button *delete* of a drawing editor produces an action that deletes the selected shapes from the current 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. Also, these standard widgets work identically in many GUI platforms. Naturally, current GUI testing frameworks rely on this concept of standard widgets and have demonstrated their ability to find errors in GUIs composed of such widgets [23], [11], [3], [21], [28].

However, the current trend in GUI design is the shift from designing GUIs composed of standard widgets to designing GUIs relying on more complex interactions[2] and *ad hoc* widgets [5], [6], [7]. By *ad hoc* widgets we mean non-standard widgets developed specifically for a GUI, such as the drawing area of graphical editors. 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*" [6]. The essential objective of this shift 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. Consequently, standard widgets are being more and more replaced by *ad hoc* ones and interactions are shifting from mono-event to multi-event interactions (*e.g.* bi-manual / multi-touch interactions for zooming, rotating, *etc.*)

[1]Also called a command [13], [5] or an event [23].

[2]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 interacts with objects in the real life.

that aim at being more adapted to users. In this paper we argue that a similar transition must occur in the GUI testing domain: move from testing standard GUIs to testing advanced GUIs by leveraging human-computer interaction breakthroughs. This transition will provide new concepts and tools to reveal the new types of errors that emerge in advanced GUIs.

Indeed, GUI model-based testing approaches relying on standard widgets show their limits for testing such new kinds of advanced GUIs. If they demonstrated their efficiency to find crashes and regressions in standard GUIs, testing advanced GUIs requires the ability to test *ad hoc* widgets and their complex interactions that existing approaches cannot test.

This paper proposes to leverage UIDLs dedicated to build models of advanced GUIs in order to drive the generation of test scenarios tailored to the detection of errors introduced in advanced GUIs. As highlighted by Cohen *et al.*, "*Models are needed for high quality system testing in GUIs, since they guide test generation and ensure sufficient coverage*" [11]. Leveraging such a model of interactions, the contributions of this paper are as follows:

- a precise analysis of the limits of current GUI model-based testing approaches to detect errors in advanced GUIs;
- the use of Malai, a UIDL to model advanced GUIs;
- a series of novel adequacy criteria that guide the selection of test scenario from the test model;
- the concept of *interaction-action-flow graph* (IFG) for going beyond the current limits of EFGs.
- initial feedback from the use of the proposed approach in an industrial project and on an open-source interactive system.

As results, we demonstrate the expressiveness limits of the current GUI models and EFGs. The use of Malai permits to go beyond such limitations and to find errors in interactive systems with tested with our approach.

The paper is organized as follows. Section II presents an illustrative example used throughout this paper to explain the current limits and our approach. Section III explains the current limitations. Section IV introduces the Malai UIDL and its benefits. Section V details the adequacy criteria and the concept of interaction-action-flow graph. Section VI highlights the benefits of our proposal through experiments and describes research challenges to overcome. This paper ends with related work (Section VII) and the conclusion presenting the future GUI testing challenges (Section VIII).

II. Illustrative Example

A. Presentation of Latexdraw

Latexdraw[3] is an open-source drawing editor for LaTeX that we selected as an illustration of interactive systems relying on more natural but complex interactions and widgets. Latexdraw is used throughout the paper to explain our approach.

Figure 2 shows Latexdraw's GUI composed of both standard widgets grouped in two toolbars, and an *ad hoc* widget corresponding to the drawing area of the editor. This drawing area is special compared to standard widgets and is representative of other *ad hoc* widgets because of the three following points: 1) Data are dynamically presented in the widget in an *ad hoc* way. In our case, data are graphically represented in 2D. 2) Data presentations can be supplemented with widgets to interact directly with the underlying data. For instance with Figure 2, the text shape "Data" is surrounded by eight scaling handlers and one rotation handler. 3) *Ad hoc* interactions are provided to interact with the data representations and their associated widgets. These interactions are usually more complex than standard widget's interactions. For instance, the scaling and rotation handlers in Latexdraw can be used with a drag-and-drop (DnD) interaction. With a multi-touch screen, one can zoom on shapes using two fingers (a bi-manual interaction) as it is typically more and more the case with mobile phones and tablets.

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

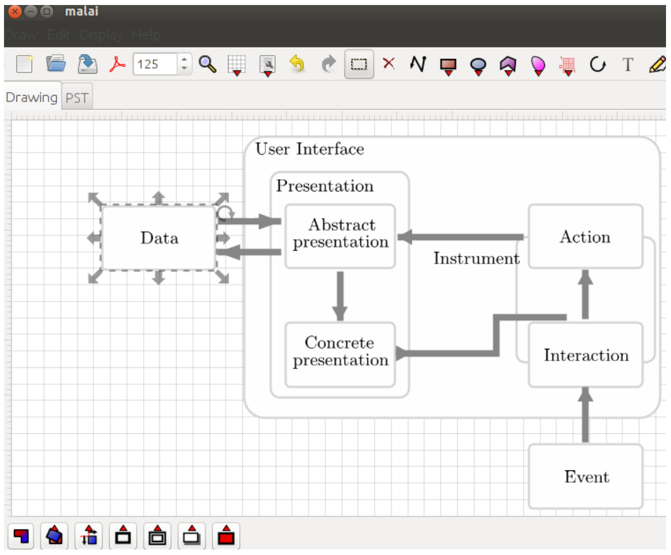


Fig. 2: Screen-shot of Latexdraw

B. Errors to find in Latexdraw

Latexdraw' GUI is composed of several standard widgets that can be tested with current GUI testing frameworks to find crashes. However, one may want to detect that executing one interaction on a GUI has the expected results. For instance, if a user performs a bi-manual interaction on the drawing area, the expected results are either a zoom action if no shape are targeted by the interaction; or a scale action applied on the targeted shape. The zoom and the scale levels are both computed from information provided by the interaction. One may want to validate that the correct action has been performed and that the execution of the action had the expected results (*e.g.* no crash or the zoom level is correct). This kind of faults consists in validating that the specifications have been implemented correctly.

III. Current Limitations in Testing Advanced GUIs

Using Latexdraw as an illustrative example, we explain in this section the limitations of the current approaches for testing advanced GUIs and thus for detecting errors such like those detailed in Section II-B.

A. Studying errors found by current GUI testing frameworks

GUITAR is one of the most widespread academic GUI testing frameworks [15]. It follows the standard GUI testing process depicted by Figure 1 and can be thus studied to highlight and explain the limitations of the current GUI testing approaches for testing advanced GUIs.

GUITAR developers provide information on 95 major errors detected by this tool during the last years in open-source interactive systems[4]. An analysis of these errors shows that: 1) all of them have been provoked by the use of standard widgets with mono-event interactions, mainly buttons and text fields; 2) all of them are crashes. While several of the tested interactive systems provide advanced interactive features, all the reported errors are related to standard widgets. For instance, ArgoUML[5] is a modeling tool having a drawing area for sketching diagrams similarly to Latexdraw. None of the 9 errors found by GUITAR on ArgoUML has been detected by interacting with this drawing area.

We applied GUITAR on Latexdraw to evaluate how it manages the mix of both standard widgets and *ad hoc* ones, *i.e.* the drawing area and its content. If standard widgets were successfully tested, no test script interacted with the drawing area. In the next section, we identify the reasons of this limit and explain what is mandatory for resolving it.

B. Limits of the current GUI testing frameworks

The main differences between standard widgets and *ad hoc* ones are: one can interact with standard widgets using a single mono-event interaction while *ad hoc* ones provide multiple and multi-event interactions; *ad hoc* widgets can contain other widgets and data representations (*e.g.* shapes or the handlers to scale shapes in the drawing area). Moreover, as previously explained, four elements are involved in the typical GUI testing process: the GUI oracle; the test model; the language used to build GUI models; the model creation process. In this section we explain how current GUI and test models hinder the ability to test advanced GUIs.

Current GUI models are not expressive enough. Languages used to build GUI models, called UIDLs, are a corner-stone in the testing process. Their expressiveness has a direct impact on the concepts that can compose a GUI test model (*e.g.* an EFG). That has, therefore, an impact on the ability of generated GUI test cases to detect various GUI errors. For instance, GUITAR uses its own UIDL that captures GUI structures (the widgets that compose a GUI and their layout).

However, in the current trend of developing highly interactive GUIs that use *ad hoc* widgets, current UIDLs used to test GUIs are no longer expressive enough. First, *UIDLs currently used for GUIs testing describe the widgets but not how to interact with them*. The reason behind this choice is that current GUI testing frameworks test standard widgets, which behavior is the same in many GUI platforms. For instance, buttons work by pressing on it using a pointing device on all GUI platforms. This choice is no more adapted for advanced GUIs that rely more and more on *ad hoc* widgets and interactions. Indeed, the behavior of these tests has been developed specially for a GUI and is thus not standard. As depicted in Figure 3, GUITAR embedds the definition of how to interact with widgets directly in the Java code of the framework. Test scripts notify the framework of the widgets to use on the SUT. The framework uses its widgets definitions to interact with them. So, supporting a new widget implies extending the framework. Even in this case, if users can interact with a widget using different interactions the framework randomly selects one of the possible interactions. Thus, the choice of the interaction to use must be clearly specified in GUI models. That will permit to generate a test model (*e.g.* an EFG) that can explore all the possible interactions instead of a single one. UIDLs must be expressive enough for expressing such interactions in GUI models.

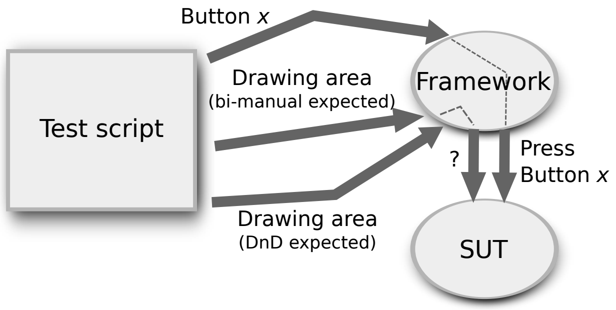


Fig. 3: Representation of how interactions are currently managed and the current limit

Second, *Current UIDLs do not support multi-event interactions*. The current trend in GUI design is to rely more on natural but complex interactions (*i.e.* multi-event interactions such as the DnD and the bi-manual interactions) rather than on mono-event ones (*e.g.* button pressure). Moreover, such interactions can be *ad hoc*. Testing such interactions implies that UIDLs must be able to specify them. Following the previous point on describing interactions in GUI models, the main benefit here would be the ability to generate from GUI models test models leveraging the definition of these interactions.

Last, *current UIDLs do not describe the result of the use of one widget on the system*. Users interact with a GUI in a given purpose. The action resulting from the interaction of a user with a GUI should be described in GUI models. Similarly, the dependencies between actions and the fact that some actions can be undone and redone should be specified as well. For instance, an action *paste* can be performed only if an action *copy* or *cut* has been already performed. That would help the creation of GUI oracles able to state whether the result of one interaction had the expected results. For instance with Latexdraw, a GUI model could specify that executing a DnD on the drawing area can move a shape at a specific position if a shape is targeted by the DnD. From this definition, a GUI oracle could check that on a DnD the shape has been moved at the expected position. Another benefit would be the ability to reduce the number of the generated test cases by leveraging the dependencies between actions. Such work has been already proposed by Cohen *et al.* [11] and must be capitalized in GUI models.

[4]https://spreadsheets0.google.com/spreadsheet/pub?key=0Ah-FLr0Egz9zdGtINk54ZDlDbWdaNWdOSzR0clB4ZkE
[5]http://argouml.tigris.org/

Current EFGs mix both interaction, widget, and action under the term *event*. Test models, *i.e.* EFGs in our case, are graphs of all the possible *events* that can be produced by interacting with a GUI. An event is both the widget used and its underlying interaction. The name of the event usually gives an indication regarding the action produced when interacting with the widget. For instance, Cohen *et al.* describe a test model example is composed of the events *Draw*, *Paste*, *Copy*, *Cut*, *etc.* corresponding to both the menu items that produce these events and their interaction, here *Menu Pressed* [11]. The name of each event describes the action resulting from the use of the widget. However, this mix of concepts may hamper the testing process of advanced GUIs as explained as follows.

First, *actions must be reified in EFGs*. Currently, GUI oracles detect crashes or regressions and are not supplemented with information coming from EFGs. However, testing advanced GUIs requires the detection of other kinds of errors as explained in Section II-B. In our previous work, we empirically identified and classified various kinds of GUI errors that can affect both standard and advanced GUIs [20]. For instance with Latexdraw, a GUI oracle must be able to state whether a shape has been correctly moved. EFGs should contain information about the actions defined in GUI models in order to provide GUI oracles with information mandatory for stating on test verdicts.

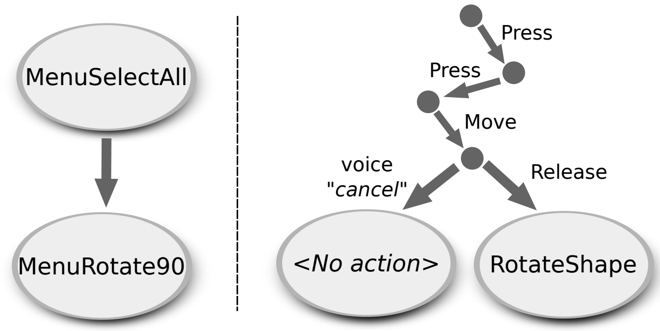


Fig. 4: EFG sequence on the left. Interaction-action sequence on the right.

Last, *interactions and widgets must be clearly separated in EFGs*. As depicted by the left part of Figure 4, mixing interaction and widget works for standard widgets that use a mono-event interaction. For testing *ad hoc* widgets using several multi-event interactions, EFGs should explicitly describe how user interactions work. For instance, the right part of Figure 4 precisely specifies the events that compose the interaction performed by a user (here a multi-touch interaction) to rotate shapes or to cancel the interaction: two touch pressures followed by a move and a touch release to rotate shapes; canceling the interaction in progress consists of pronouncing the word "cancel". In this case, EFGs would be able to support *ad hoc* interactions.

We explained in this section the precise limits of the current GUI model-based testing approaches for testing advanced GUIs. In the following section we propose to leverage the UIDL Malai to tackle the limits of GUI models. In Section V we introduce our testing approach to tackle the limits of EFGs. Section VI present experiments we conducted with our approach to find errors described in Section II-B in Latexdraw and in the context of an industrial project. We then describe research challenges to overcome for testing advanced GUI using model-based testing techniques.

IV. Modeling GUIs with Malai

Malai is an architectural design pattern that aims at fitting the growing evolutions of software interactivity [7], [8]. Malai gathers principles of several major interaction models and design patterns, notably the *instrumental interaction* [5], the *direct manipulation* [31], and the *Command* and *Memento* design patterns [13]. Compared to the model-view-controller (MVC) architectural design pattern [19], Malai refines the controller to consider the notion of interactions, actions, and instruments as first-class objects. With dedicated tools, Malai can be used as a UIDL.

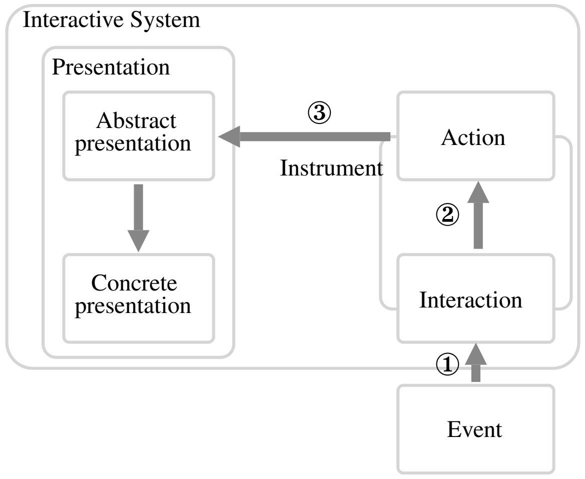


Fig. 5: The architectural design pattern Malai

Malai decomposes an interactive system as a set of presentations and instruments (see Figure 5). A presentation is composed of an abstract presentation and a concrete presentation. An abstract presentation is the data model of the system (the *model* in MVC). A concrete presentation is a graphical representation of an abstract presentation (the *view* in MVC).

An action encapsulates what users can modify in the system. For instance, Latexdraw has numerous actions such as *rotating shapes*. An action does not specify how users have to interact with the system to perform it. Action just specifies the results of a user interaction on the system. An action can also depend on other actions for being executed. For example, the action *paste* can be executed only if an action *copy* or *cut* has been executed before. An action is executed on a presentation (link ③).

An interaction is represented by a finite-state machine (FSM) where each transition corresponds to an event produced by an input device (Figure 5, link ①). Using FSMs for defining interactions permits the conception of structured multi-event interactions. An interaction is independent of any interactive system that may use it. For instance, a bi-manual interaction, as depicted by Figure 6, is defined as an FSM using events

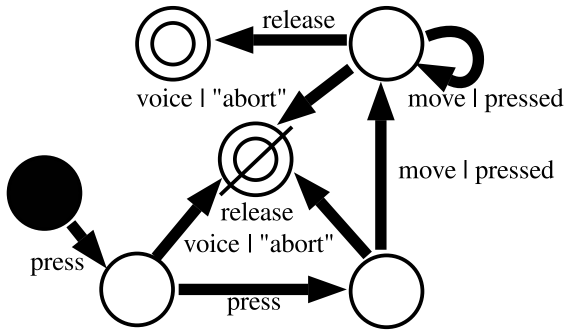


Fig. 6: Example of a Malai interaction: a bi-manual interaction

produced by pointing devices or speech recognizers (*e.g.* pressure, voice). This interaction does not specify the actions to perform on a system when executed (*e.g.* rotate shapes). The interaction depicted by Figure 6 starts at the initial state (black circle) and ends when entering a terminal (double-lined circle) or an aborting (crossed out circle) state. Aborting states permit users to abort the interaction they are performing. Transitions (*e.g. press* and *move*) can be supplemented with a condition constraining the trigger of the transition. For instance, the interaction goes into the aborting state thanks to the transition *voice* only if the pronounced word is "*abort*".

Because actions and interactions are independently defined, the role of instruments is to transform input interactions into output actions (link ②). Instruments reify the concept of tool one uses in her every day life to manipulate objects [5]. For instance, Figure 7 describes the instrument *Hand* as it could have been defined in Latexdraw. The goal of this instrument is to move and rotate shapes. Performing these actions requires different interactions: rotating shapes can be done using the bi-manual interaction previously depicted; moving shapes can be done using a DnD interaction. In Malai such tuples interaction-action are called *interactors* and one instrument can have several interactors, *i.e.* one can handle an instrument with different interactions to execute different actions. In an interactor, the execution of the action is constrained by a condition. For example, the interactor *Bimanual2Rotate* (Figure 7) permits the execution of the action *RotateShapes* only if the source and target objects of the bi-manual interaction are the same shape: $src == tgt \wedge src\ is\ Shape$ (using a bi-manual interaction to rotate a shape may imply that the two pressures are done on the targeted shape).

In Malai, the GUIs of a system are composed of the widgets provided by all the instruments: an interaction can be based on a widget (*e.g.* a click on a button) and in this case, instruments using such interactions create and provide these widgets.

Is Malai expressive enough for modeling advanced GUIs in a testing purpose? Yes, it is: Malai supports mono-event as well as multi-event interactions; interactions work as FSM that may help the test model generation process; actions can be defined and dependencies between them can be specified; instruments and their interactors can be viewed as FSMs as well.

Related UIDLs. There exists other UIDLs that can be used to model advanced GUIs, such as *Interactive Cooperative objects* (ICOs) [26]. The ICOs architecture uses Petri nets to model advanced user interactions. UML can also be used to model user interactions using its state machine diagrams [29].

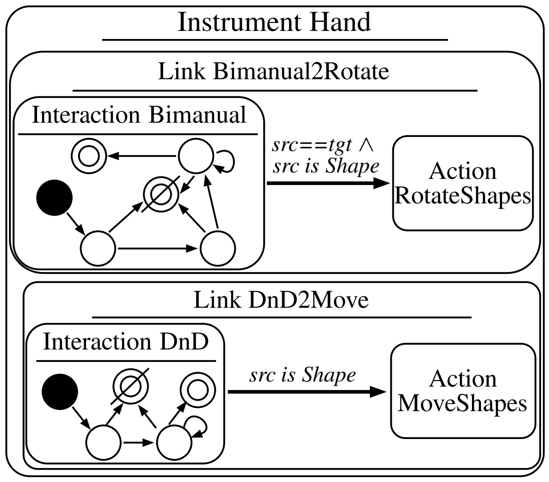


Fig. 7: Illustration of a Malai instrument

V. GUI testing with Malai

This section details how Malai can be leveraged to test advanced GUIs. First, novel adequacy criteria focusing on user interactions rather than on widgets are introduced. Then, the concept of interaction-action-flow graph is explained. Finally, the composition of test scripts is detailed.

A. Adequacy criteria

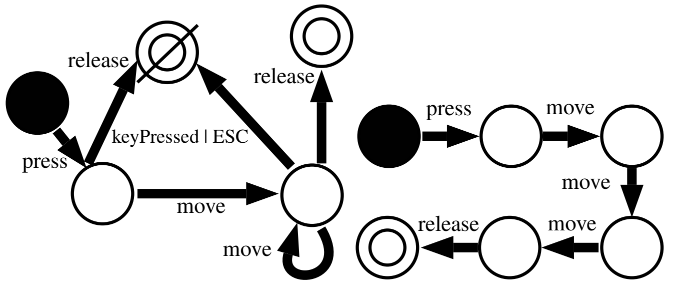
One of the contributions of this work is to leverage the Malai UIDL to define novel adequacy criteria. The novelty of the proposed adequacy criteria is to focus on interactions, modeled in Malai as FSMs, and on instruments which convert interactions into actions. The definition of the adequacy criteria based on interactions and instruments requires the definition of the following concepts.

Definition 1 (Interaction Path): Given an interaction modeled by an FSM, an *interaction path* is a path of the FSM going from the initial state to one of the ending states.

An interaction path corresponds to one execution of an interaction. For example, Figure 8a describes the FSM of a DnD interaction. The basic behavior of a DnD starts by a pressure (using a mouse or a touch screen) followed by at least one move and ends with a release. This interaction can be aborted if the key *escape* is pressed or if a release occurs before a move. One interaction path of this interaction (Figure 8b) is a pressure followed by three moves ending with a release.

Definition 2 (Interactor Path): Given an interactor of an instrument, an *interactor path* is composed of an interaction path followed by the action that may be produced constrained by one solution of the condition.

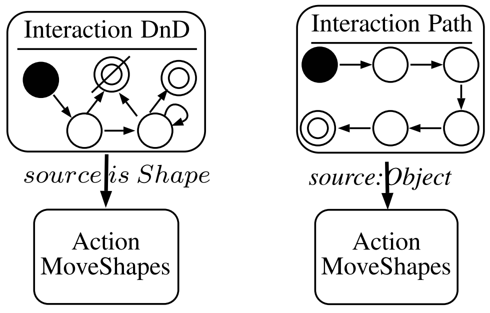
An interactor path is produced each time a user interacts, *i.e.* executes an interaction, with the system to perform an action, successfully or not. For example, Figure 9a represents an interactor that maps a DnD interaction to an action that



(a) A drag-and-drop interaction (b) A drag-and-drop path

Fig. 8: An interaction and one of its interaction paths

moves shapes. The condition to execute the action states that the source object targeted by the DnD must be a shape: *source is Shape*. One possible interactor path of this interactor (Figure 9b) is the interaction path described in Figure 8b where the source is not a *shape* but an instance of *Object*. In this case the action will not be executed.



(a) An interactor (b) An interactor path

Fig. 9: An interactor and one of its interactor paths

Definition 3 (Instrument Path): Given a set of instruments where each instrument contains a set of interactors, an *instrument path* is an ordered sequence of interactor paths.

Usually, users handle several instruments to perform an ordered sequence of actions to reach their initial goal. Such a sequence is an instrument path. For example, Figure 10 describes three instruments composed of interactors. The instrument *Hand* (the two interactors ③) has already been detailed in Figure 7. The instrument *Pencil* (interactor ①) permits the creation of shapes by doing a DnD. The condition to execute this action is that the source and target points of the DnD must not be equal (otherwise a shape would have an invalid height and width). The instrument *EditingSelector* (interactor ②) permits selecting between the *Hand* or the *Pencil* by clicking on their corresponding button (*handB* or *penB*). One possible instrument path is depicted in Figure 11 showing a sequence of interactor paths.

Based on these definitions, the interaction adequacy criterion and the instrument adequacy criterion can be defined as follows.

Definition 4 (Interaction adequacy criterion): A set TCp of test cases satisfies the *interaction adequacy criterion* if and only if for all interaction transitions t (*i.e.* input events), there is at least one test case tc in TCp such that tc triggers transition t.

Definition 5 (Instrument adequacy criterion): A set TCi of test cases satisfies the *instrument adequacy criterion* if and only if for all interactor paths l, there is at least one test case tc in TCi such that tc triggers l.

Based on these adequacy criteria, a GUI model-driven test cases generator algorithm has been defined. The goal of this algorithm is to generate abstract test cases covering these adequacy criteria.

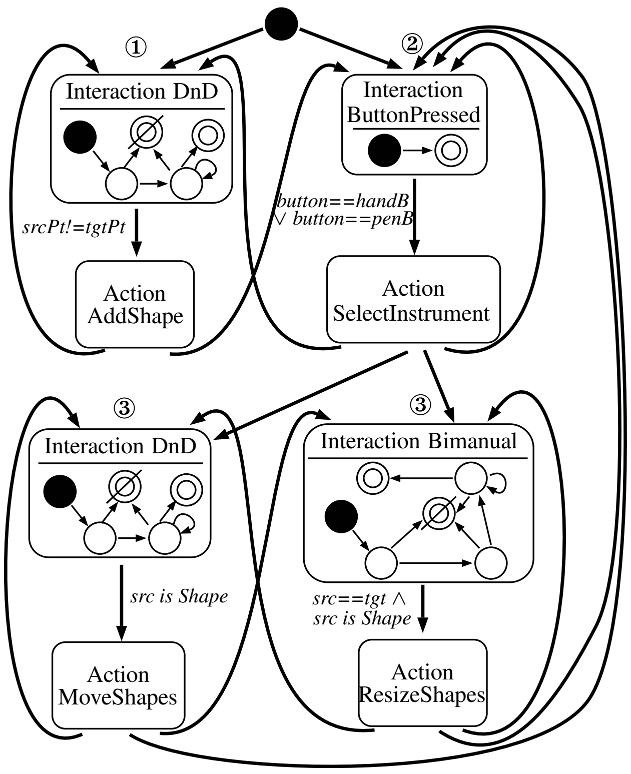


Fig. 10: Example of an interaction-action-flow graph

B. *Interaction-action-flow graph*

This section introduces the concept of interaction-action-flow graph (IFG) that are automatically inferred from Malai models. As depicted in Figure 10, an IFG follows the same idea than an EFG by sequencing all the possible user interactions. The difference is that the concepts of interaction, action, and widget are clearly separated, and interactions and actions are included in IFGs. The goal of such separation is to be able to test a widget using its different interaction, and to test that the effective result of one execution of an interaction is the result expected ass defined in the action. So, an IFG is a sequence of interaction-to-action tuples. Each tuple is in fact an FSM composed of two states (the interaction and the action) and one transition (the condition that permit to execute the action). For instance, Figure 10 describes an IFG composed of the three instruments described in the previous section. At start-up, only the instruments *Pencil* and *EditingSelector* are activated and can be used. So, the interaction-action tuples ① and ② are available. The instrument *EditingSelector* (②) permits to use either the *Pencil* or the *Hand* instrument. So, using this instrument can lead to all the tuples ①, ②, and ③. Figure 10 does not represent the widgets associated to each tuple but the tuples ① and ③ deal with the drawing area while the tuple ② deals with a set of buttons.

To create such sequences from Malai models, we need to define which instruments can be used at a given instant. Malai models do not explicitly specify the relations between instruments, *i.e.* they do not specify which instruments can be handled after having used an instrument. Such relations are mandatory to obtain from Malai models a graph of instruments from which test scripts can be produced.

Inferring these relations is automatically performed by analyzing the Malai actions: there exists actions that activate or inactivate instruments. For instance with Figure 10, the instrument *EditingSelector* is dedicated to select (*i.e.* to activate/inactivate) between the instruments *Hand* or *Pencil*. Thus, after having used the *EditingSelector* instrument, either the instruments *Hand* or *Pencil* can be used (transitions ①). Similarly, after having used the instrument *Hand*, respectively *Pencil*, the instrument *EditingSelector* can be used only (*Pencil*, respectively *Hand*, is inactivated, transitions ②).

C. Test script

In our approach, a test script is an ordered sequence of interactor paths (see Definition 2). For instance, Figure 11 illustrates the composition of a test script with the example of the drawing editor previously introduced. The depicted test script excerpt shows an initial interactor path composed of an interaction path leading to the execution of the action *SelectInstrument* when the button used is *handB*. Following, another interactor path is defined containing another interaction path leading to the action *MoveShapes*. The length of this sequence is defined before the generation of the test cases.

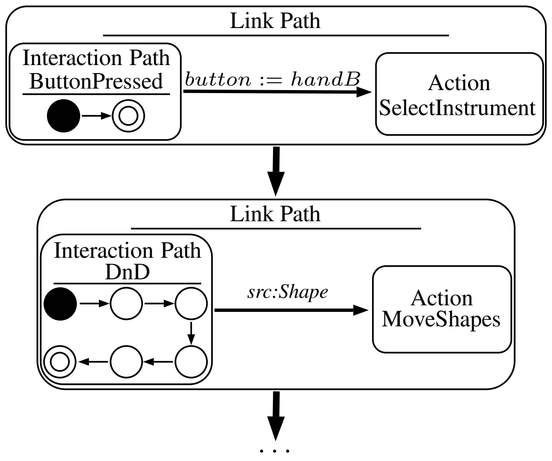


Fig. 11: Example of a test script composed of a sequence of interactor paths

VI. Experiments

To validate the expected benefits of the use of Malai and the concept of IFG, we applied our method on two different use cases:

1) **Latexdraw** (3.0-alpha1): an open-source vector drawing program for LaTeX, introduced in Section II;
2) **the CONNEXION project** : an industrial project that aims at improving model-based GUI testing techniques.

We conclude this section by discussing about the threats to validity we identified during our experiments.

A. Case study 2: latexdraw

1) Presentation: As explained previously, Latexdraw is a vector drawing editor for LaTeX written in Java and Scala (see Figure 2). Latexdraw has been selected for our evaluation for three reasons. First, Latexdraw is developed by one of the authors of this paper so that its behavior is mastered for doing experiments. Second, as explained previously Latexdraw has an *ad hoc* widget, the drawing area, that can be used with different advanced interactions. Finally, Latexdraw is broadly used since it has been downloaded more than 2.300 times per month since 2011 (according to the sourceforge's statistical tool).

The goal of this experiment is not to provide benchmarks against GUITAR or other automated GUI testing tools. Indeed, GUITAR is a fully automated framework while we manually built Malai models and manually executed the test scripts. The goal of this experiment is to show that, independently of the automation of the testing process, the current GUI testing approaches have intrinsic limits because of being build on the top of the concept of standard widget.

2) Testing Latexdraw: We started by manually modeling the interactive system using Malai. Because of resource limitations we do not modeled the whole system but selected several major actions and instruments. We modeled 24 instruments containing 71 interactors. Concerning actions, 37 actions have been modeled doing various tasks such as exporting the current drawing or moving selected shapes. Then, an IFG has been automatically generated. We parameterized the abstract test case generator using the Instrument adequacy criterion: all the instruments must be tested at least one time. As a result, 142 abstract test cases have been generated. We then manually concretized and executed these abstract tests on the SUT. These tests have been manually executed since out tool does not support Java Swing yet (it has been initially developed for the next use case).

3) Results and discussion: Table I gives an overview of the defects, described in the following sections, found during our experiments. The manual execution of the tests led to the detection of 4 defects. All of them were not reported in the official bug trackers of Latexdraw. These defects concern different parts of the system.

Defects #1 and #2 have detected that several user interactions did not work as expected. These user interactions are a multi-click and a DnD interactions. Both of them are *ad hoc* interactions developed specifically for Latexdraw. With a multi-click interaction, users can click is several locations in the drawing area to create a shape composed of several points. In our case, executing the interaction did not perform the excepted action, *i.e.* the creation of the shape. The DnD interaction can be aborted. It means that while executing the DnD, the user can press the key "*escape*" to stop the interaction

Systems \ Defects	Id	Link	Origin
Latexdraw	#1	https://bugs.launchpad.net/latexdraw/+bug/788712	Interaction
	#2	https://bugs.launchpad.net/latexdraw/+bug/768344	Interaction
	#3	https://bugs.launchpad.net/latexdraw/+bug/770726	data model
	#4	https://bugs.launchpad.net/latexdraw/+bug/770723	Java Swing

TABLE I: Defects found by applying the proposed approach on the tested systems

and to not execute the associated action. These two defects cannot be find by GUITAR since: they have been provoked by non-standard interactions; the two interactions can be executed on the drawing area and we explained that GUITAR cannot support multiple interactions for a same widget.

Defect #3 was an issue in the data model where changes in the Latexdraw's preferences were not considered.

Defect #4 was not a Latexdraw issue but a Java Swing one: clicking on the zoom buttons had not effect if performed too quickly. Still, this defect has been detected thanks to the action that zooms on and out the drawing area.

B. Model-based testing GUI in the industry: feedback from the CONNEXION project

1) Presentation: The CONNEXION[6] project is a national industry-driven work program to prepare the design and implementation of the next generation of instrumentation and control (I&C) systems for nuclear power plants. One goal of this project is the development of model-based techniques for testing GUIs of such I&C systems, more precisely for:

- automating as far as possible the current error-prone, expensive, and manual GUI testing process;
- finding GUI errors as early as possible in the development phases to reduce the development cost;
- finding various kinds of GUI errors (not only crashes) on various kinds of *ad hoc* widgets.

The main constraint, imposed by norms [17], [18], is that the testing process of I&C systems must be driven from their specifications. So, producing entire GUI models using reverse engineering techniques from the SUT is not possible (some specific information, however, can be extracted from GUIs to be used in GUI models as discussed below).

2) Testing Process: Similarly to the previous case study (*i.e.* Latexdraw), we manually design GUI models from the provided specifications using Malai. We then automatically generate abstract GUI tests. These last are then concretized to produce executable GUI tests that run on top of the *SCADE LifeCycle Generic Qualified Testing Environment* (SCADE QTE), a tool for testing GUIs developed with the SCADE Display technologies[7]. The concretization phase includes two steps:

- Mapping abstract test cases to the targeted testing framework and SUT. This step requires manual operations to map GUI model elements to their correspond elements in the GUI under test. For instance, the name of the widgets of the GUI under test have to be mapped to the widgets defined in the GUI models. To avoid this step, the specifications have to precisely specify the names to use, as discussed later in this section.
- Generating GUI oracles. We currently generate basic oracles that checks the correct GUI workflow when interacting with widgets.

3) Challenges to overcome: If the Malai expressiveness permits to test advanced GUIs, we face the following GUI testing challenges relative to the concretization phase:

GUI oracles generation. In our previous work, we identified and classified standard errors that affect GUIs [20]. Based on this classification, we aim at automatically producing GUI oracles that are able to find these various kinds of errors. These GUI oracles have to be produced as far as possible from GUI models. GUI errors, however, can take many forms. It implies the development of multiple GUI oracles based on techniques diametrically different. For instance, the graphical nature of GUIs requires their graphical rendering to be checked. To do so, a possible oracle consists of comparing screenshots of GUIs to detect differences. This oracle thus uses image processing techniques. However, checking that a widget is correctly activated can be done using classical code unit testing techniques. These differences between GUI oracles complexifies the GUI model-based testing process that has to generate such oracles.

Incomplete, ambiguous GUI specifications. The specifications we use to build GUI models do not formally specify in detail the GUI to test. As a result, the GUIs and the GUI models, produced independently each other from the specifications, may differ. For instance, the position of widgets are not explicitly precised. Testers and developers have to determine these positions from GUI mockups. To limit this problem in the CONNEXION project, some information are extracted from the developed GUIs to be reused in the GUI testing models. These extracted information are notably the position of the widgets. It means that the position of the widgets is not tested but is used to detect other GUI errors (*e.g.* widget responsiveness). This problem of incomplete, ambiguous GUI specifications has been already reported in previous work on model-based testing. Dalal *et al.*, for instance, reported that "*some of the constraints were not covered in the requirements and we had to run the application to see exactly how it had been implemented*" [12].

C. Discussion

The first results of our experiments highlight the capability of our approach to detect defects in advanced GUIs. We were able to test multiple *ad hoc* interactions used on advanced widgets. Actions permitted to compare the effective result of

[6]http://www.cluster-connexion.fr/
[7]http://www.esterel-technologies.com/

interacting with SUTs and the expected one. However, we face several challenges that currently hinder efficient GUI model-based testing processes. First, the complexity and the diversity of GUI oracles, due to the various natures of GUI errors, require the development of multiple techniques for generating GUI oracles from GUI models. Second, the ambiguous and incomplete aspects of GUI specifications hamper the model-based testing process.

Our next challenges will be the automation of the process including the automatic generation and execution of test scripts that implies the generation of GUI oracles being able to detect errors other than crashes.

VII. Related Work

Several challenges have been faced to provide an effective GUI test cases generation. Current solutions have adopted model-based testing (MBT) process to build models describing SUTs. We identified several studies focusing on MBT for GUI testing that: build models from specifications [36],[34] or SUTs [22]; apply different formalisms to represent models (*e.g.* state machines [30], graph models [23], formal languages [9]); combine a range of tools to provide automated GUI testing (*e.g.* GUITAR [15], TERESA [25]).

Nevertheless, current approaches exploit MBT to generate test cases for GUIs based on standard widgets and mono-events instead of multi-interactions from advanced input devices. Thus, to our best knowledge all the solutions to automate GUI testing do not address advanced GUIs. Memon *et al.* [24] and Yuan *et al.* [35] propose different adequacy criteria for GUI testing. If the relevance of these criteria has been demonstrated for testing standard GUIs, our work propose new kinds of criteria dedicated to multi-event interactions over advanced GUIs.

The following approaches use an MBT process for GUI testing by considering standard widgets. For example, Memon [23] has proposed the framework GUITAR to automate GUI testing based on a reverse engineering process. GUITAR captures the GUI structure of the SUT by extracting widgets and their properties to then transforms the GUI structure into an event-flow graph. Each node represents a widget event (*e.g.* click on button open) and its reachability is a possible test script. Thus, EFGs permit generating test cases to cover all the sequence of GUI events. However, part of the test cases are not feasible since they do not take into account the dependencies between events. Thus, in [16][3], the authors leveraged GUITAR EFG model to provide test cases more optimized.

Huang *et al.* [16] applied a genetic algorithm to identify infeasible test cases in EFG and firing an exception when they are replayed. Arl *et al.* [3] improved the test suite by building an event dependency graph that contains dependencies extracted from the widgets event handlers in bytecode. In [2], the test suite is parameterized with addition of input data to widgets. Even though these approaches aims at refining test suite generated by the EFG, they are still based on mono-event performed on standard widgets.

Silva *et al.* [32] presented a solution to automate the oracle testing by adopting the task models as oracle. The ability of task models to describe GUIs behavior (*e.g.* input and output tasks) motivated this adoption. However, the process is partially automated considering that the final oracle has to be refined manually by a tester to add extra information. The solution is limited to test the expected user behavior. To improve this solution Barbosa *et al.* [4] modified the original task model by introducing users errors. Nevertheless, the task models are limited to represent simple GUI interactions since these tasks can not represent the design of the system and its complete behavior.

Nguyen *et al.* [27] proposed an approach that permits to build models of the SUT by separating the business logic from the presentation logic. The solution is based on the *Action-Event Framework* [28] that uses the AEFMAP language to build the mapping model. The mapping links the abstraction actions (represented by action model) to GUI events. Based on that mapping model, the presentation logic test cases are converted automatically from the business logic test cases. The main idea behind the two layers proposed by the authors is to ease the reuse through the action model when changeable GUIs must be designed. However, we achieve in our approach a higher level of abstraction by generating abstract test cases based on Malai which separates the abstract presentation from concrete presentation.

Mariani *et al.* [21] presented the AutoBlackTest technique to build a model generating test cases incrementally and automatically while interacting with a GUI of SUTs. While interacting with the SUT, the GUI is analyzed to extract its current state. Then, the behavioral model is updated to select and execute the next action. Thus, the model is built incrementally and the test cases are identified. Those test cases are refined and a test suite is then generated automatically. The solution is limited to interactive applications at the GUI level by supporting only standard widgets. In contrast with our approach, the solution does not leverage abstract test cases to increase the level of abstraction since the set of generated test cases represents the concrete test cases in MBT process and still needs to be refined before executing the test suite.

Bowen and Reeves [10] proposed a method to generate abstract tests from formal models by adopting the test-first development approach. The solution builds models from the functional and UI requirements. To cover the entire system, these models are mapped into a presentation relation model. From these models, the abstract cases are generated automatically and the concrete tests are created. Although the solution is close to our approach by using an MBT process to build models and generate abstract cases for interactive systems, it still focuses on testing GUIs by manipulating only standard widgets, their properties and behaviors.

Mobile computing is an interesting domain in which users can interact with applications leveraging new kind of interactions (*e.g.* touch, multi-touch screen). Several approaches explore MBT to build models reusable on other hardware platforms. For example, GUITAR has been extended to support the Android platform [1]. Although, this platform enables multi-interaction, the solution is limited to perform single interactions (*e.g.* tap) over the Android device emulator.

Takala *et al.* [33] presented a solution to build models of Android applications and generate online GUI test cases. The solution uses two separate state machines to build the model to facilitate its reuse on other device models. The first machine represents the state of the SUT and verify it against the model. The second state machine uses the keyword method to capture the user interactions (*e.g.* tapping and dragging objects on the screen). However, the solution is fully dependent on the Android platform and can not be applied on other domains.

EXSYST uses evolutionary algorithms to improve GUI test suites by maximizing code coverage [14]. Even if this approach works with standard widgets, we think that it can be extended to support advanced ones using our proposal.

VIII. Conclusion and Research Agenda

This paper details the current limits of GUI testing frameworks for testing advanced GUIs, and introduces a new approach for testing such GUIs. We leverage the Malai UIDL to define GUI models more expressive than the current ones. We then propose the new concept of interaction-action-flow graph extending the concept of even-flow graph. We applied our approach on 2 interactive systems to validate the ability of our approach to reveal defects typical of advanced GUIs that the current GUI testing frameworks cannot detect. One major challenge to target in our future work is the generation of GUI oracles being able to detect defects other than crashes.

Acknowledgements

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

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of android applications. In *Proc. of ASE'12*, pages 258–261, 2012.

[2] S. Arlt, P. Borromeo, M. Schäf, and A. Podelski. Parameterized GUI tests. In *Proc of Testing Software and Systems*, pages 247–262. 2012.

[3] 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.

[4] A. Barbosa, A. C. Paiva, and J. C. Campos. Test case generation from mutated task models. In *Proc. of EICS'11*, pages 175–184. ACM, 2011.

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

[6] M. Beaudouin-Lafon. Designing interaction, not interfaces. In *Proc. of AVI'04*, 2004.

[7] A. Blouin and O. Beaudoux. Improving modularity and usability of interactive systems with Malai. In *Proc. of EICS'10*, pages 115–124, 2010.

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

[9] J. Bowen and S. Reeves. UI-design driven model-based testing. *ECEASST*, 22, 2009.

[10] J. Bowen and S. Reeves. UI-driven test-first development of interactive systems. In *Proc. of EICS'11*, pages 165–174. ACM, 2011.

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

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of ICSE'99*, pages 285–294. ACM, 1999.

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

[14] F. Gross, G. Fraser, and A. Zeller. Exsyst: search-based gui testing. In *Proc. of ICSE'12*, pages 1423–1426, 2012.

[15] D. R. Hackner and A. M. Memon. Test case generator for GUITAR. In *Companion of ICSE'08*, pages 959–960. ACM, 2008.

[16] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. pages 245–254, 2010.

[17] IEC. Nuclear power plants - main control-room - verification and validation of design, 1995.

[18] IEC. Nuclear power plants - control rooms - design, 2010.

[19] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in smalltalk80 system. *Journal of Object Oriented Programming*, 1:26–49, 1988.

[20] V. Lelli, A. Blouin, and B. Baudry. Classifying and qualifying GUI defects. In *Proc. of ICST'15*. IEEE, 2015.

[21] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *Proc. of ICST'12*, pages 81–90, 2012.

[22] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proc. of the WCRE'03*, page 260, 2003.

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

[24] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proc. of FSE'01*, pages 256–267. ACM, 2001.

[25] G. Mori, F. Paterno, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Eng.*, 30(8):507–520, Aug. 2004.

[26] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.*, 16(4):1–56, 2009.

[27] D. H. Nguyen, P. Strooper, and J. G. Suess. Model-based testing of multiple GUI variants using the GUI test generator. In *Proc. of the 5th Workshop on Automation of Software Test*, pages 24–30. ACM, 2010.

[28] D. H. Nguyen, P. Strooper, and J. G. Süß. Automated functionality testing through GUIs. In *Proc. of the 33th Australasian Conference on Computer Science*, pages 153–162, 2010.

[29] OMG. UML 2.1.1 Specification, 2007.

[30] R. Shehady and D. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers*, pages 80–88, 1997.

[31] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[32] J. L. Silva, J. C. Campos, and A. C. R. Paiva. Model-based user interface testing with spec explorer and ConcurTaskTrees. *Electron. Notes Theor. Comput. Sci.*, 208:77–93, 2008.

[33] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an android application. In *Proc. of ICST'11*, ICST '11, pages 377–386. IEEE Computer Society, 2011.

[34] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1), Feb. 2007.

[35] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE Trans. Software Eng.*, 37(4):559–574, 2011.

[36] T. Yue, S. Ali, and L. Briand. Automated transition from use cases to UML state machines to support state-based testing. In *Proc. of ECMFA'11*, pages 115–131. Springer-Verlag, 2011.