Rationalizing the Need of Architecture-Driven Testing of Interactive Systems

Alexandre Canny[1(✉)], Elodie Bouzekri[1], Célia Martinie[1], and Philippe Palanque[1,2]

[1] ICS-IRIT, Université Paul Sabatier – Toulouse III, Toulouse, France
{alexandre.canny, elodie.bouzekri, celia.martinie, philippe.palanque}@irit.fr
[2] Department of Industrial Design, Technical University Eindhoven, Eindhoven, Netherlands

Abstract. Testing interactive systems is known to be a complex task that cannot be exhaustive. Indeed, the infinite number of combination of user input and the complexity of information presentation exceed the practical limits of exhaustive and analytical approach to testing [31]. Most interactive software testing techniques are produced by applying and tuning techniques from the field of software testing to try to address the specificities of interactive applications. When some elements cannot be taken into account by the software testing technique, they are usually ignored. In this paper we propose to follow an opposite approach, starting from a generic architecture for interactive systems (including both software and hardware elements) for identifying in a systematic way, testing problems and testing needs. This architecture-driven approach makes it possible to identify how software testing knowledge and techniques can support interactive systems testing but also where the interactive systems engineering community should invest in order to test their idiosyncrasies too.

Keywords: Architecture-driven testing · Interactive system testing

1 Introduction

Interactive systems testing involves different methods and techniques depending on the objectives of these tests. The field of Human-Computer Interaction (HCI) has been focusing on finding defects that affect user-related properties (such as usability, user experience, accessibility, learnability…) developing methods, techniques and tools to perform user studies involving directly the end-users. The field of software engineering has been focusing on finding defects that affect software quality and software-related properties (such as reliability, performance, availability, security…). This field has been developing methods, techniques and tools to perform software studies involving the Application Under Test (AUT) and a list of input to be provided to reveal defects.

Detecting defects in interactive systems requires bringing these two research fields together in order to ensure that, on one side the interactive systems fit with the human capabilities and the work of the users and, on the other side that the interactive systems are correct and behave as expected at any time. Unfortunately, the software engineering

Published by Springer Nature Switzerland AG 2019
C. Bogdan et al. (Eds.): HCSE 2018, LNCS 11262, pp. 164–186, 2019.
https://doi.org/10.1007/978-3-030-05909-5_10

field has mainly been addressing interactive systems as a standard computing system (for instance abstracting away input and output devices, interaction techniques etc.) and only seminal work from Memon in his PhD [29] was dealing with specific aspects of interactive application testing. More precisely that work was performing testing using events on interactors of WIMP applications (called Graphical User Interface (GUI) testing). However, that work (and what was done later on) remained focused on WIMP interfaces [39] while the field of HCI has been proposing much more efficient and complex interaction techniques (e.g. the survey on menu techniques in [5]). More recently, research work on software programming of interactive applications [25] proposed methods and tools to automatically reveal bad programming practices but this covers only a very small part of the interactive software (the event-handlers and their structuring). What is tested and what is not, is a critical point as, if testing only some parts of the interactive system might reveal defects, the non-tested parts might still jeopardize the actual use of the system. Some recent work has been trying to extend the part of the AUT beyond the GUI by considering the execution platform (e.g. Android [38]). However, even in that work, testing only involves testing via event-handlers, thus remaining close to GUI testing aspects.

In this paper we propose to follow an approach starting from a generic architecture dedicated to interactive systems (including both software and hardware elements) for identifying, in a systematic way, testing problems and testing needs. This architecture-driven approach makes it possible to identify how software testing knowledge and techniques can support interactive systems testing. It also allows identifying where the interactive systems engineering community should invest to design and develop testing techniques complementary to the software engineering ones.

Section 2 introduces informally some of the testing problems that are specific to interactive systems. It demonstrates that those problems span from hardware (input and output devices) to the functional core (non-interactive) of the application. It thus demonstrates the need for testing techniques dedicated to interactive systems. This section also identifies testing principles that could (and should) be applied to support testing activities to address these problems of interactive systems. Section 3 presents in detail the MIODMIT generic architecture for interactive systems and positions the testing problems presented in Sect. 2 on that architecture. Section 4 presents two case studies and the testing problems they raise to make concrete the abstract problems presented in Sect. 2. These case studies exhibit different interaction techniques and different input devices highlighting the variability of interactive systems and how this affects the testing needs. The generic architecture MIODMIT is tuned for each case study and is used to systematically identify those testing needs. Section 5 connects MIODMIT to human aspects thus positioning user testing together with interactive system testing presented in previous sections. Section 6 structures the related work presented in the paper and makes explicit the testing problems that are covered by the literature and the remaining open ones. Section 7 concludes the paper and highlights paths for future work.

2 Informal Description of Problems for Testing Interactive Systems

Since interactive systems relies on a growing set of I/O devices to enable interaction, it is important to look at the testing of both their hardware and their software components. In this section, we present the main principles of software, hardware and usability testing and then use these principles to exemplify some of the problems tester must take into account when testing interactive systems.

2.1 Main Principles of Testing

Main Principles of Software Testing
Software testing is an activity every application should go through, no matter it is interactive or not. The Software Engineering Body of Knowledge (SWEBOK) [11] defines software testing as the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.

A key point in the software testing activity is the definition of the test levels. The test level of an application is defined thanks to two variables: the target of the test and the objective of the test. The targets of testing can be a single module (unit testing), a group of module (integration testing) or the entire software application (system testing). The SWEBOK [11] references 12 objectives of testing such as performance testing or regression testing which are respectively non-functional and functional testing techniques. The non-functional tests refer to the way the software operates (e.g. is it to slow?) whereas the functional tests refer to the extent to which the software behaves properly (e.g. is it producing the correct output for a given set of input?).

Once the testing level is defined, the testing of software application requires three activities: the test case construction, the test suite construction and the test execution. During the testing activity, the tested software is usually referred as the Application Under Test (AUT). In [31], Nguyen et al. details these steps considering the testing of applications with a graphical user interface (GUI) using "standard" widgets (e.g. buttons, label, radio button).

Main Principles of Hardware Testing
The testing of the hardware of interactive systems remains, to the best of our knowledge, a relatively unexplored area. While patents such as [20] proposes testing techniques for testing touch screen at hardware level, no systematic classification of testing requirements for hardware has been issued to specifically address interactive systems. However, hardware testing is a concern in the field of Cyber-Physical System (CPS) engineering that shares some specificities with interactive system engineering.

CPS integrate both physical and computational elements so their engineering requires bridging the continuous analog real world and the discrete digital world. The behavior of CPS is thus similar to the one of modern interactive with multiple I/O devices (e.g. compass, camera, speaker, haptic devices). According to Asadollah et al. [3], hardware testing consists in testing hardware components of CPS, including tests

of each component's functionality, which descriptions are based on the system requirements. Amongst the most common and important variable in testing CPS hardware, Asadollah et al. [3] lists memory size, speed, storage capacity, I/O interfaces (ports), synchronization capabilities, etc. They also point out that testing the hardware under specific conditions (e.g. local environment) is required. For interactive systems, this is equivalent to the testing of the interactive system in its context of operation.

It is important to note that as for software testing, test levels may be defined. Hardware tests levels can be described using targets and objectives. For example, testing a touchscreen on its own is comparable to performing unit testing while testing an entire smartphone packing this touchscreen is comparable to performing system testing.

The tested hardware may be referred as the System Under Test (SUT) even though this expression is also used in software engineering. In this paper, we consider that:

- the **AUT** is the application running on the interactive system at testing time. For example, on a Personal Computer where VLC Media Player is running for a video playback, the AUT is the VLC Media Player;
- the **SUT** is the entire interactive system, including its Input/Output devices, drivers, etc. During a video playback with VLC Media Player, the SUT thus includes the screen, the speakers, the soundcard, the remote control (if any), the operating system, the VLC Media Player application, etc.

2.2 Testing Interactive System

To highlight how interactive system testing is difficult, we present in this section some informal examples of the diversity of requirements and constraints that have to be taken into account while testing (Table 1). These examples find their origins in the definitions of elements of interactive systems (e.g. modal window), in the specifications of interactive systems (e.g. hardware capability) or in authors' experiences with interactive systems (e.g. text disappearing or mouse cursor not moving in Windows).

On interactive systems, it must for instance be tested that if multi-touch interactions involve five fingers, the touchscreen must accommodate at least five fingers. This requirement appears in Table 1 (H3) as "The I/O devices must comply with the requirements for the I/O devices of the SUT". Second column of Table 1 assigns a name to each example that will be used later. The third column assigns to each example a component of interactive systems that is involved in this requirement/constraint. This column shows that our examples of requirements and constraints (to be tested on interactive systems) requires testing both software (e.g. Non-Interactive Code, Interactive Code) and hardware (e.g. device) components of an interactive system. However, to the best of our knowledge, no integrated testing techniques offer support for the entire interactive system. A review of the literature regarding "interactive system testing" shows that these keywords link mostly to papers related to GUI Testing. Banerjee et al. [6] define GUI testing to mean that a GUI-based application, i.e. one that has a graphical user interface (GUI) front-end, is tested solely by performing sequences of events (e.g. "click on button", "enter text", "open menu") on GUI widgets (e.g. "button", "text-field", "pull-down-menu"). Thus, hardware is not took into account in

Table 1. Examples of the diversity of requirements and constraints to be tested

Description	Name	Component
Unit testing of the software components that are responsible of providing data and services for the AUT should not reveal defects	N1	Non Interactive Code
Integration of the software components that are responsible of providing data and services of the AUT with the interactive elements of the interactive system should not reveal defects	N2	
A modal window reduces the interaction space only until it is closed	I1	Interactive Code
The position of the manipulator of an input device (e.g. pointer) should evolve in accordance with user action on that device (e.g. mouse pointer going left if the mouse is moved to the left)	I2	
The user can only trigger authorized events (e.g. whenever a file is open the user can trigger the event close file)	D1	
The user can trigger none of the unauthorized events (e.g. a user cannot trigger the event close file if the file is not open)	D2	
The text within a button must always remain visible when the button is visible	P1	Presentation
The grayed-out widgets should not produce event even though the user act on them	P2	
Every available widgets should be reachable (e.g. if the widget is not visible, manipulation of its window should allow the user to make it visible)	P3	
The I/O loop should have performance compatible with human perception (e.g. the movement of the manipulator of the mouse should occur within 50 ms after the mouse has been moved)	H1	Device
The color displayed on the screen should correspond to the one that has been required to be displayed	H2	
The I/O devices must comply with the requirements for the I/O devices of the SUT (e.g. the touchscreen device should handle at least as many fingers as what is needed by the SUT)	H3	
The I/O devices must behave so they prevent undesired repetition of events and produce expected repetition of events (e.g. keeping a key pressed on the keyboard will repeat production of the event associated to that key)	H4	
The behavior of the firmware of the input device should be compatible (e.g. type of data, rate) with the one of the input device driver	D3	Driver
The SUT should prevent removal of needed I/O devices by an application if another application requires access to it (e.g. if a microphone is required all the time by an application (noise detection), another one will not be allowed to access it)	C1	Input/Output Manager
The SUT must be capable of producing high-level events from low-level events that are produced by one input device (e.g. each time the user presses and releases a mouse button, a mouse clicked event is produced in addition to mouse up and mouse down events)	C2	
The SUT must be capable of exploiting multiple output modalities synchronously if the AUT needs it (e.g. sound+video during video playback)	C3	
The SUT must be capable of producing high-level events from low-level events that are produced by multiple input devices (e.g. moving two fingers concurrently in the opposite direction should be interpretable as a pinch event)	C4	

GUI testing. Moreover, as mentioned by [26], GUI testing do not scale properly with advanced GUI (e.g. supporting multi-touch or multimodal interaction). We claim that a better understanding of the role of interactive systems components may help to provide better testing techniques for interactive and so we propose to work on architecture-driven testing techniques.

3 Architecture of Interactive Systems and Its Use for Testing

Interactive systems testing is a complex activity that is only partly addressed by existing testing approaches. Indeed, a review of the literature regarding interactive system testing shows that most of the problems presented in the Sect. 2.2 are not addressed. Most of existing testing techniques [27, 28] focus on GUI (Graphical User Interface) involving standard UI widgets (e.g. Buttons, ComboBox). As GUI heavily exploits the functionalities, the interactive objects and the input devices offered by the underlying execution platform, testing approaches rely on the "good" functioning of the platform and thus testing only addresses behavioral and functional aspects of the application and not the interactive system as a whole. In order to perform a systematic approach to testing we propose an architecture-driven testing for interactive systems. In order to avoid the pitfalls of GUI testing we propose to include hardware and hardware drivers in addition to the more standard software elements. In this section, we first present some architectures for interactive systems and highlight their relevance for identifying components to test. Then, we detail the MIODMIT architecture, a fine-grained architecture covering in a comprehensive way all the elements of "modern" interactive systems. Finally, we highlight the components of the architecture impacted by the problems presented in Sect. 2.2 and describe the testing needs to prevent these problems.

3.1 Interactive Systems Architectures

Since the early 80's, a software architecture (known as the Seeheim model [34]) has been proposed to decompose interactive applications in smaller components. To reflect the evolution of interactive applications and the fact that a large share of application code was devoted to the User Interface, Len Bass et al. [7] proposed the Arch model that was decomposing Seeheim's Presentation component into two, the Logical Interaction component and the Physical Interaction one.

With that modification, 3 out of 5 components are dedicated to the UI and the Physical Interaction component connects to input and output devices (even though not explicitly mentioned in the paper). Beyond that, it does not cope explicitly with multimodal UIs that are nowadays mainstream means of interaction.

The architecture associated to ICon [16] refines carefully the input flow from physical input devices to the application core (see. Fig. 5.1. in [17], p. 148), however, no description of the output management is provided. As interactions frequently involve both perception and action dimensions of user behavior, refining only input does not make it possible to describe real systems.

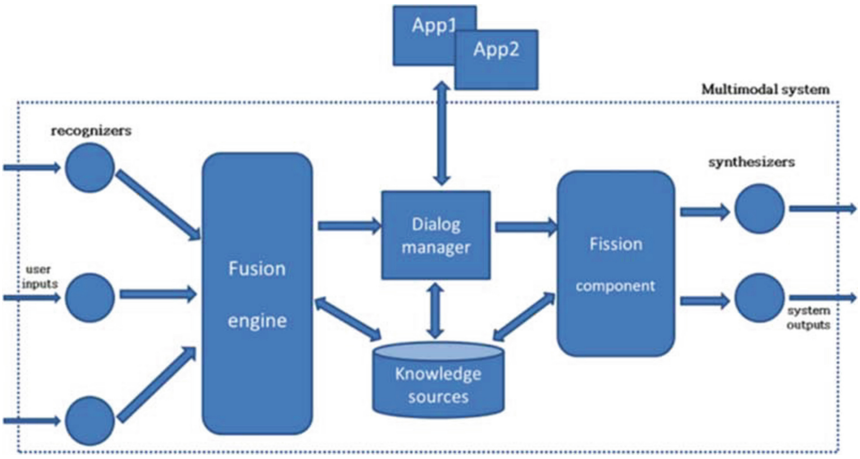

Fig. 1. Architecture for multimodal interactive systems from [15]

As stated above, current interactive applications exploit multiple input and output devices, and interaction with those systems may be multimodal. Some contributions such as [15] (see Fig. 1) present connected components forming an architecture of multimodal interactive applications. This architecture makes explicit a fission component (for output) and a fusion component for input. Such representation is misleading as fusion and fission may be associated with both input and output. For instance, a sentence produced by a speech recognition system might be broken down into words (fission of the input information) to identify commands and parameters [23]. Similarly, a presentation of information might require the combination of multiple information (e.g. the level of danger and a warning message) in order to present fused information to the user (e.g. the warning message in orange color). This demonstrates the importance of having a very detailed and generic architecture for describing and designing interactive systems.

3.2 MIODMIT Generic Architecture for Interactive Systems

MIODMIT [13] is a detailed architecture that makes the interactive systems components explicit including hardware ones (both for input and output). It is thus more representative about interactive systems that the other architectures presented in the related work section. This architecture does not exhibit dedicated fusion or fission engines components, as fusion and fission are functions are distributed over the architecture in several components (explained more in detail in the case study section). Figure 2 presents an overview of the MIODMIT architecture. Each rounded rectangle represents a component of the MIODMIT architecture and arcs represent the communication between component. When an arc between two components is not present, the component cannot communicate (information flow, function call, …) with the other one.

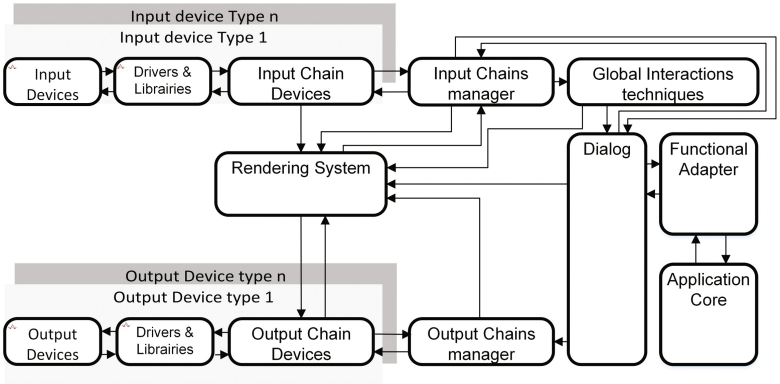

Fig. 2. The MIODMIT architecture (adapted from [14])

The "Input Device Type" greyed-out box describes the information flow for a given type of input device. Each new type of input device requires a separate "Input Device Type". An "Input Device Type" is composed of three components. First, "Input Devices" component is the physical (hardware) input device manipulated by the user (e.g. a mouse or a finger on a touchscreen). The "Input Devices" component sends information to or receives requests of information from the "Drivers & Libraries" software component, which, in turn, makes this information available to the other components of MIODMIT. Less commonly, "Drivers and Libraries" can manage "Input Devices" behaviour such as sampling frequency [24] or providing user identification [40]. "Drivers and Libraries" can be provided either by the "Input Devices" manufacturer or by the operating system if the hardware is standard or has been around for a significant amount of time. Lastly, the "Input Chain Device" component is a software component that mirrors the state of the "Input Devices" hardware (called "Virtual Device"), the "Logical device" of the "Input Devices" hardware (e.g. cursor pointer position for a mouse) and a manager. These components are transducers [2] that transform raw data into low-level information. Virtual device can be dynamically instantiated with plug-and-play devices. Whereas, logical devices can be dynamically instantiated at operation time. For example, each time a finger touches a multi-touch input device, a new logical device associated with the new finger is created. The manager addresses configuration and dynamic configuration of devices.

The "Input Chain Manager" component is an event-based component that processes low-level information and connects such information to user interface objects (e.g. a button) and their location on the screen. This component may fuse information from different input devices to create high-level information (e.g. clicking simultaneously on two mice will produce one click on each and the "Input Chain Manager" might produce higher-level event called "combined click" [1]). In addition, this component manages dynamic reconfiguration of interaction in case of failure[1].

[1] As terminology for failures, faults and errors we use the definitions from [4].

The “Input Chain Manager” component sends high-level information to “Global Interaction Technique” component (a transducer [2]) or “Dialogue” component or to the both.

The “Global Interaction Technique” component is a transducer that performs a recognition of a specific interaction technique, which is not linked with a user interface object (e.g. “OK Google” vocal interaction). Moreover, this component generates high-level information used by the application to trigger the various command it provides.

The “Dialogue” and “Core” components are similar components to standard interactive systems architecture such as Seeheim [34] or Arch [7].

The “Rendering System” component is responsible of immediate feedback and other state-based rendering functions. A state-based rendering function describes how to present information of a specific state.

The “Output Chain Manager” component offers same functionalities as the “Input Chain Manager” component. Nevertheless, the “Output Chain Manager” is state-based whereas the “Input Chain Manager” is event-based.

The “Output Device Type” describes the information flow for output device in the same way.

3.3 Locating Testing Problems and Testing Needs Using MIODMIT

In this section, we position the problem listed in Table 1 according to the MIODMIT architecture. This systematic analysis highlights the fact that testing problems may be related to various components of the interactive systems and that a precise description of the interactive system is required to be able to manage all these problems. It is important to note that while some problems affect only one component of the architecture, some of them are distributed over several.

Problems Related to a Single Component of MIODMIT

N1 - Unit testing of the software components that are responsible of providing data and services for the AUT should not reveal defects and N2 - Integration of the software components that are responsible of providing data and services of the AUT with the interactive elements of the interactive system should not reveal defects

In interactive system architecture in general, as well as in MIODMIT, the “software components that are responsible of providing data and services” are part of the **application core**. The testing of these components is well-documented by the software testing community and the techniques for testing the application core of an interactive application are not different from those allowing the testing of non-interactive applications.

I1 - A modal window reduces the interaction space only until it is closed

Modal windows are designed so they force the user to interact with them before they can resume interaction with their parent applications. Thus, *I1* means that developers must verify that any way of closing the modal window will allow user to resume interaction according to their choice within the modal window. This implies testing at the **rendering system** level.

P2 - The grayed-out widgets should not produce events even though the user acts on them
This means that even though the **input chain manager** produces a mouse clicked event over a grayed out button, testing should prove that this event should not be forwarded as a higher-level event produced by the button itself towards other components of the application (such as the dialogue).

H3 - The I/O devices must comply with the requirements for the I/O devices of the SUT
This means that the compliance of every **Input/Output device** with their specifications must be verified before their integration in the interactive system.

C1 - The SUT should prevent removal of needed I/O devices by an application if another application requires access to it
This means that testing the component responsible of dynamic reconfiguration of the I/O in the **input/output chain manager** must be performed in order to ensure that this component will not cause a loss of resource for an application.

C2 and C4 - The SUT must be capable of producing high-level events from low-level events that are produced by one input device/The SUT must be capable of producing high-level events from low-level events that are produced by multiple input devices
This means that the capability of the **input chain device** to produce high-level events specific to a device (e.g. click) must be tested (C2). Moreover, the capability of the **input chain manager** to produce high-level events from the events produced by **input chain devices** must be tested (C4).

C3 - The SUT must be capable of exploiting multiple output modalities synchronously if the AUT needs it
This means that the priority management of the **output chain manager** must be tested.

Problems Distributed Over Several Components of MIODMIT
I2 - The position of the manipulator of an input device (mouse cursor) should evolve in accordance with user action on that device

This problem concerns the entire left part of the architecture, or short loop (input device types, rendering system and output device types). To take it into account, a proper transcription of the user input on the output device is required. For a mouse, this means that:

- Its motion sensor is calibrated properly (**input device**);
- Its **drivers and libraries** are getting data consistently and are computing the mouse acceleration properly;
- The **input chain device** produces high level event notifying the **rendering system** of the new cursor location;
- The **rendering system** makes the proper rendering request to the **output chain device** (including coordinates, shape of the mouse cursor, etc.);
- The **output chain device** combines rendering request from the **rendering system** and the **output chains manager** so the cursor is always drawn of top;
- The **output drivers and libraries** are dispatching the rendering requests to the graphic card properly (correct screen, resolution, etc.)

- The screen (**output device**) is set in the proper input (e.g. HDMI) and is capable of displaying the cursor.

P1 - The text within a button must always remain visible when the button is visible
This means that the **output chain manager** must request the display of the button with the text in it and that the output chain device behave as expected. We do not detail the testing needs for the **output device type** (presented in problem *I2*).

D1 and D2 - The user can only trigger authorized events/The user can trigger none of the unauthorized event
This means that the rendering of the application produced by the **output chains manager** and the **output device type** should reveal which actions are authorized or not (e.g. disabling widgets) and that the problem *P2* has been taken into account.

H1 – The I/O loop should have performance compatible with human perception
This problem is a refinement of problem *I2* that takes human performance into account. The I/O device type and the computing system responsible for the rendering system must be performant enough so they accomplish the whole behavior described in *I2* in an acceptable time regarding human perception.

H2 - The color displayed on the screen should correspond to the one that has been required to be displayed
This means that the **output chains manager** and **output chain type** must only request the display of colors the **output device** is capable to render. Moreover, the screen (**output device**) must be calibrated for its targeted color space (e.g. RGB) and the **drivers and libraries** must be configured properly so they use the screen's color space.

H4 - The I/O devices must behave so they prevent undesired repetition of events and produce expected repetition of events
This means that, at the hardware level (**input/output device**), proper implementation of feature such as de-bouncing must be verified (e.g. for a keyboard **input device**). This also mean, at the **input device type** level, the implementation of character repeat is done properly.

D3 - The behavior of the firmware of the input device should be with the one of the input device driver
This means that the **input device** should always produce information that can be used properly by the **drivers and libraries**. Thus, if the **drivers and libraries** and/or the firmware of the **input device** is/are updated, both elements must still be compatible.

4 Testing Interactive Systems: Two Cases Studies

In this section, we present how to use MIODMIT to identify the testing needs for two different MS Windows interactive systems

- a version of the application designed to be used with a mouse, a keyboard and a trackpad as input devices and a screen as output device;

- a multimodal version of the application with the same input/output devices and adding multi-touch input and speech-recognition. Besides, this application uses a loudspeaker and speech synthesis.

4.1 A Common Application Core for Both Cases Studies

Both case studies are drawing applications that allow manipulating drawings (i.e. creation of colored shapes selected from a finite set of possible shapes and colors). This allows the two applications to share the same Application Core, i.e. the component that is responsible for maintaining a list of created shapes, their color and their position. Since the applications are coded in JAVA, unit testing of the application core is possible using tools such as JUnit. Such testing allow verifying that:

- The services provided by the Shape class (e.g. getColor(), setColor(Color c), getPosition()) behave as expected;
- The class responsible for handling the current drawing behaves as expected (e.g. addShape(Shape s), getNbShape());
- The class responsible of serializing and de-serializing drawings behave as expected (e.g. open(File f), save(), saveAs(File f)).

It is important to note that the testing of all this services independently is however insufficient. Indeed, the internal behavior of the class must also be assessed with respect to user action e.g. the user cannot open a file already open, save an empty file, etc.

4.2 Case Study 1: Mouse, Keyboard and Screen

Informal Description of the Interactive System and Its Architecture
In this first case study, the interactive system specifications are the following: HP Zbook, Operating System: Windows 10; Output device: 14 in. display 1920 * 1080; Input devices: Pointing devices (Integrated trackpad and HID-compliant USB Mouse) and Integrated Keyboard.

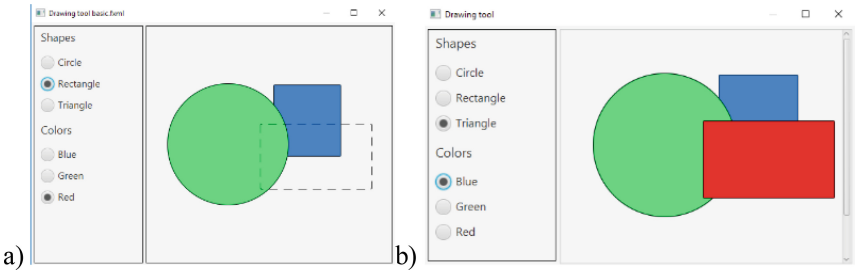

Fig. 3. Screenshots of the interactive application (a) after drawing two shapes and while drawing a third one (b) after drawing four shapes (including one not visible, please notice the scrollbar) (Color figure online)

The user can select the shape and the color by clicking on the associated radio buttons (see Fig. 3). To position the shape in the drawing area, the user has to press the left button of the mouse at the desired location of the first point of a rectangle containing the shape. Maintaining the mouse left button down (dragging) until the desired shape size creates a ghost (Fig. 3a). Releasing the left button adds the shape to the drawing.

Following the «tune-on-demand» process presented in [14], we can produce from MIODMIT a specific architecture (see Fig. 4). The two "pointing device type" are the mouse and the trackpad. The "Mouse Device" and "Keyboard device" represent the hardware part of these input devices. The "Mouse Driver" and "Trackpad Driver" represent the drivers of these input devices. Similarly, the "keyboard device type" is described by the "keyboard device" and "keyboard driver". The "output device type" corresponds to the computer screen composed of a "screen device" and a "screen driver". As computer runs Windows 10, the "Windows Manager" of this Operating System covers entirely the functions of input and output chain components as well as a subset of the functions of the rendering component. The "Windows Manager" contains the "Abstract Cursor" (input chain functions and rendering functions) and the "Feedback Cursor" (output chain functions and the rendering of the cursor). The "Dialogue" component describes the behavior of the interactive application. The "Functional Core" supports the functions presented in Sect. 4.1.

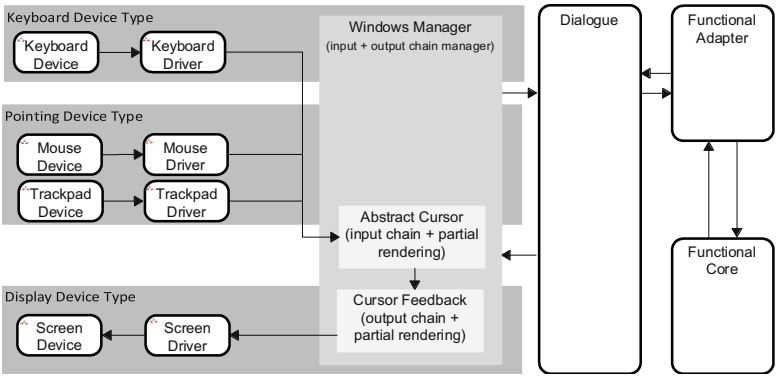

Fig. 4. Description of the interactive system using MIODMIT

Systematic Identification of Testing Needs for the Interactive System

This section identifies testing needs exploiting Fig. 4 from right to left (functional core testing needs are omitted as they were presented in Sect. 4.1.

Dialogue of this application must handle discrete events from GUI widgets (i.e. radio button) and events produced in the drawing area. Testing the capabilities of the **dialogue** in handling events from GUI widgets is actually the main objective of most GUI testing techniques. Indeed, in [29], techniques are designed to test GUI driven by mono-event interactions (e.g. button clicks) which are not suitable for multi-event interaction (e.g. on the drawing area of a graphical editor). While [10, 26] discuss the testing of interactive systems with continuous interaction, these papers mainly

addresses multimodality itself and thus do not contribute to dialogue testing. Testing the **dialogue** requires:

- Verifying that it is capable of consuming all the events produced by all the input device types that it must support;
- Verifying that its user-driven state changes only occur in response to authorized events;
- Its transition between states occur as expected.

While developing in JAVA, the operating system and the JAVA Virtual Machine share responsibility over the **Windows Manager**. This component encapsulates the **Input/Output Type/Chain Device/Chain Manager** as well as the **Rendering System** according to MIODMIT terminology. For this reason, testing of the **Window Manager** thus cannot be placed under the responsibility of the developer of the application. However, the actual behavior of the **Windows Manager** raises problems that testers must take into account. Indeed, in the presentation of this case study, we state "the user can select the shape and the color by clicking on the associated radio buttons". However, the radio button is a component from the JAVA Swing library and its standard behavior does not comply with this statement. Indeed, pressing "Space" or "Enter" on a focused radio button would trigger the same "ActionEvent" as the one produced when clicking on it, adding unspecified behaviors to the application. During development, decisions regarding these unspecified behaviors must be made (should they be prevented or not?) so the test cases and suites are prepared accordingly:

- Actions described in the application specifications trigger ActionEvents as required;
- ActionEvents can (or cannot) be triggered by shortcut/hotkeys whether is was (or was not) decided to allow them in the application.

By default, the **Windows Manager** allows users to resize and move the application window. This makes it possible to hide some of the GUI widgets (e.g. Fig. 5c) or some area of the drawing (e.g. Fig. 5a and b: absence of a scrollbar does not give a proper idea of the drawing zone size). Moreover, the **Windows Manager** controls windows arrangement and focus. Testing is thus required to verify that:

- The resizing of the window is constrained enough so none of the six radio buttons are hidden;
- Resizing the window below the size of the drawing area triggers the appearance of scrollbars (e.g. Fig. 3b);
- The application window can receive focus and may or may not be visible.

The **Pointing Device Type** of this interactive system is specific as it contains two input devices: a mouse and a trackpad. The tester must verify that the abstract cursor (and its associated feedback) of the **Rendering System** encapsulated in the **Windows Manager** is capable of handling input from multiple pointing devices and is configured to do so. Indeed, on some interactive systems, each pointing device can be attached to a specific cursor[2] or can be merged in a single cursor (e.g. MS Windows).

[2] https://wiki.archlinux.org/index.php/Multi-pointer_X.

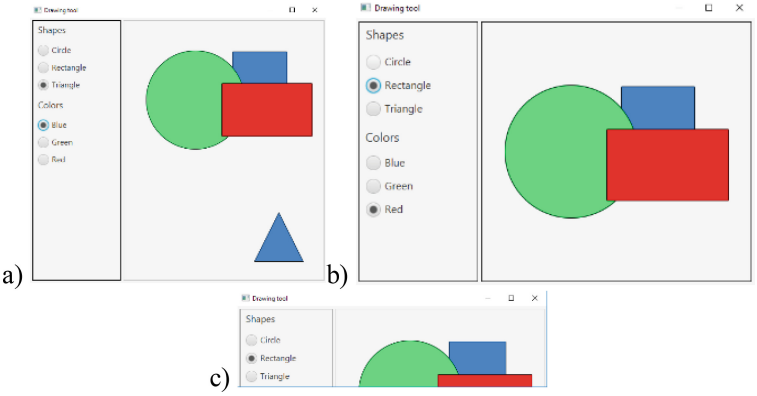

Fig. 5. The application window is (a) extended in height, (b) reduced in height after drawing the blue triangle and (c) reduced in height so radio buttons disappear (Color figure online)

The testing needs regarding the **drivers** and the **Input/Output Devices** of this interactive system were already discussed in Sect. 3.3 (see problems D3 for **drivers** and H2, H3, H4 for **I/O Devices**) and are not repeated here.

4.3 Case Study 2: The Multimodal Drawing Application

Informal Description of the Interactive System and Its Architecture

In this case study, the user can perform the same interaction as in the case study 1, in addition to multimodal ones. Since a touchscreen is available, the user can select radio buttons by touching them and can also draw a shape by sliding a finger in the drawing area. Shapes can be resized using a "pinch" interaction. The user can use a combination of voice and touch to select shape and color from existing shapes in the drawing area:

- Saying "Select this color" and then touching a shape selects the color of the touched shape and a speech synthesis announce "color selected" as in [8];
- Saying "Select this shape" and then touching a shape selects the shape of the touched shape and a speech synthesis announce "shape selected";

In both case, the touch must occur less than 2 s after speaking, otherwise the interaction is discarded.

As for case study 1, we tuned (see Fig. 6) MIODMIT generic architecture using the tune-on-demand process presented in [14]. Input/Output device types have been added as required and the multimodal aspect of the application is handled by the "Input Chains Manager" component (top right of Fig. 6). Part of the behavior of this component is implemented by MS Windows 10, while part has to be programed specifically.

Testing Needs Specific to This Multimodal Interactive System

In this section, we only present the testing needs raised by the multimodal interaction. Testing needs from case study 1 (related to mouse and keyboard interactions) remain.

The **Input Chains Manager** component introduced in this case study is a new source of event for the **Dialogue** as well as a new consumer of events from the **Windows Manager**. A key aspect in testing the **Input Chains Manager** is to support temporal aspects are required. For instance, the fusion mechanism [23] produces the selection event only if the succession of events (speech + touch) occurs within a given temporal window. It is important to note that this interaction technique has to be tested as part of the **Input Chains Manager** as it describes how some events from different input chains are produced and then transferred to the dialogue.

These case studies show that the instantiation of the MIODMIT architecture for a SUT (System Under Test) provides support for a precise identification of testing needs. It provides support in identifying:

- The common components from an application to another (in order to communalize some tests and avoid duplicated testing)
- The impact of a change in the interaction technique on the testing needs

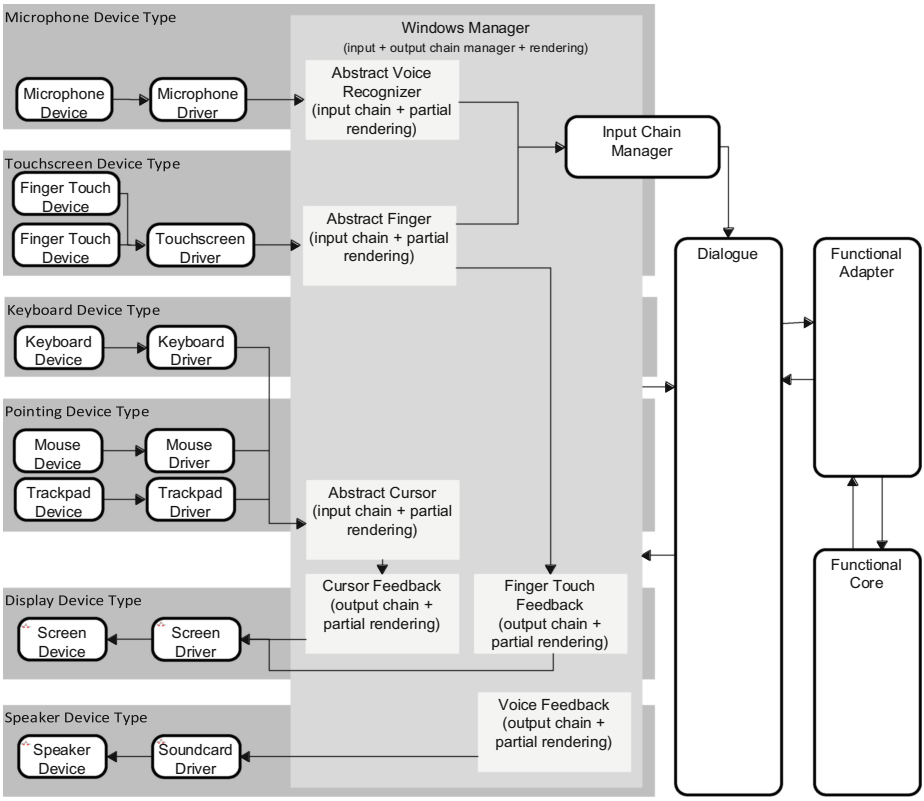

Fig. 6. Architecture of the second case study using MIODMIT

5 Human Aspects in Architecture-Driven Interactive System Testing

MIODMIT provides support for systematically testing all the parts of an interactive system. The focus of the presented work is on the interactive system side. However, while focusing on system, the human cannot be ignored. Testing some or all of the parts of the architecture of an interactive system by function and without taking into account how the future user of the system will use it, belongs to the category of system centric testing or system/software testing. For this category of testing, system/software functions are tested one by one, without caring about how they will be manipulated by the user. But, this category of testing does not take into account human aspects. An interactive system is used by a human in order to perform her/his work. The interactive system has to be at least usable by the users that are targeted for the developed system. User testing aims at taking into account human aspects for the interactive system being developed. Nevertheless, user testing increases the number of testing activities as it requires to add test cases that are related to human capabilities (colours that can be perceived by the targeted user type, font size…). At the same time, it may also decrease the number of test cases because of the limitations of human capabilities (speed human information processing, field of view…).

We argue that taking into account the human aspects when testing an interactive system is compatible with architecture driven testing. Figure 7 provides an overview of the integration of the human characteristics with the MIODMIT architecture. The following paragraph discusses the complementarity of the user testing practices with an architecture driven testing.

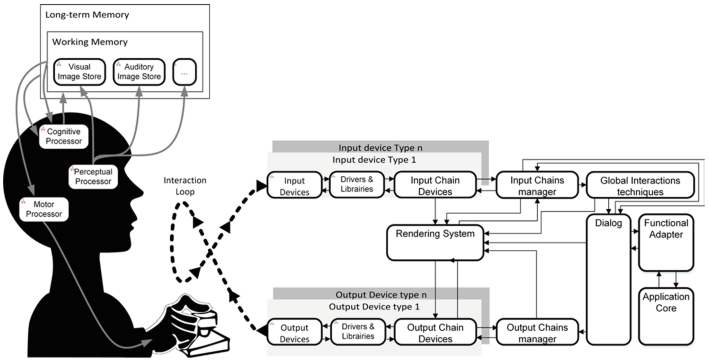

Fig. 7. H-MIODMIT (integrating the human characteristics with MIODMIT architecture)

Several properties related to human aspects may be targeted for an interactive system. The usability property is one of the most important [32]. As defined by the ISO 9241-11 [21], usability is "*the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*". Another important property that may be assessed

for an interactive system, is the learnability property, that can be evaluated by measuring how the interactive system allows the user to reach a reasonable level of usage proficiency in a short period of time [32]. Then, depending on the application domain (entertainment, games, critical systems…), other properties may also be targeted (user experience, dependability…). For example, evaluating user experience with an interactive system aims at measuring properties such as emotion, aesthetics, social connectedness [35]… that may be highly subjective and require specific evaluation processes and techniques. Several aspects of user testing for interactive systems have to be taken into account for the integration of human characteristics with architecture driven testing:

- Iterative evaluation processes are applied to ensure that the user needs are taken into account. Such processes are part of the User Centered Design paradigm which usually encompass several prototyping and evaluation phases [18]. User involvement is a pillar of such processes. User tests are performed for the most possible stages of design of an interactive system (from early design phases with low-fidelity prototypes to the deployment of the interactive system).
- For some properties, user testing can be achieved through predictive measurement (analytical techniques) and does not require direct user involvement. For example, heuristic evaluation technique [32] is based on usability principles that can be examined systematically by a usability expert, in order to detect usability issues for an interactive system. Other example is techniques based on user tasks analysis. Some of these techniques are based on user task descriptions [13, 19], and some of them are based on task models [12]. These techniques provide support to detect usability problems related to the effectiveness criterion.
- For some other properties, user testing requires user involvement (empirical techniques). For example, the wizard of Oz technique [22] is an experimental simulation performed with users. It aims at testing the interactive system by giving the impression to the user that s/he is interacting with the real interactive system. This technique can be used in the early phases of the design process, when the interactive system is partly implemented, in order to refine user needs. Another examples of testing techniques that mandatory requires user involvement are the fine tuning of an interaction technique [33] and the field user testing [36]. For that purpose, several users of the targeted user type are required to perform a limited set of task with a specific setup (part of the final interactive system, specific input device…). These techniques aim at collecting data and at analyzing performance issues (efficiency criterion) and/or subjective metrics (satisfaction, emotion, aesthetics…).

These aspects of user testing have an impact on the required level of fidelity of the interactive system under test, and thus on the precision of the description of the behavior and of the architecture of the interactive system under test:

- For some evaluation techniques, mock ups or low fidelity prototypes are sufficient to perform user testing (example of such technique is the Wizard of Oz [22]). The problem is then to ensure that the results and recommendations for the next iteration of the interactive system are feasible according to the technical constraints. The

architecture can here be of great help in providing support to filter and adapt the modifications and adding that are proposed for the future versions of the prototypes.

- For other evaluation techniques, a high-fidelity prototype or even the functional and fully or partly deployed interactive system is required (example of such technique is field user testing [36]). These kind of user testing techniques are expensive as they require to develop and setup an experimentation protocol, and to select and recruit a large number of users. In order to avoid loss of time for the users and/or loss of data for the evaluation experts, the interactive system has to function as specified and should be exempt of defects. In that case, the architecture can also be of great help by providing support to ensure that each part of the interactive system is functioning as specified before user testing. Furthermore, if the analysis of the evaluation highlights that changes are required, the architecture provides support for identifying in which parts the changes have to be performed (locality of the modifications) and thus enables to decrease the impact of these modifications and associated non-regression testing on the whole interactive system (e.g. to modify an input device driver to adapt the sensed speed of movement).

In summary, architecture-driven testing exploiting MIODMIT provides support for user testing whatever evaluation technique is used. More precisely, it supports assessing properties related to human aspects such as usability and user experience. Lastly, we highlighted the fact that even though architecture-based interactive system testing is emphasizing the technology aspect of testing, it is compliant with user centered approaches focusing on user activities and behavior.

6 Related Work

This paper presented how an architecture-driven approach can help identifying the testing needs for interactive systems. We emphasized that some of these needs are partially covered by existing testing techniques while some other are, to the best of our knowledge, not considered. In this section, we present why most of the existing techniques fail in addressing testing components of interactive systems.

First of all, we remind that the testing of the **Application Core** is similar to the testing of non-interactive applications. Due to space constraints, we do not present here the wide-range of techniques for testing non interactive-application.

GUI testing is, at the software level, the closest field to interactive systems testing. Banerjee et al. [6] systematic mapping classified 136 articles on this topic 7 years ago. Despite its age, the findings of this mapping are still relevant; especially regarding the research question “What test data generation approaches have been proposed?”. This mapping reveals that models are the most popular test generation methods in the field. While models used for test generation are, from HCI point of view, descriptions of the **dialogue** (see Fig. 3 from [30]: this figure describes the dialogue without naming it), they are not used to test the dialogue as a single component. Indeed, model-based testing tools mostly rely on the state of GUI widgets during testing. Thus, what is tested is that the presentation matches the expected state derived from the dialogue description, not that the dialogue itself is in a correct state (so it might not take into account

events from some correctly enabled GUI widgets). [6] distinguishes two other popular test generation techniques: random and capture/replay. Since the random approach is designed for "crash-testing" technique (i.e. events are played randomly on the GUI widgets to verify that the application does not crashes), they cover very partially the dialogue and functional core of the application by revealing they present defects that causes crashes. These techniques do not however reveal the source of the defect. Capture/replay is a technique in which testers records actions on the GUI that are stored to be replayed on the SUT. This is particularly useful for regression testing. However, these techniques addresses only a fraction of the output chain manager and rendering systems. Indeed, recording all the possible actions on both SUT and AUT is an impossible task as soon as one action can be performed several times on the GUI. Due to space constraints, we do not go exhaustively over all the papers presented in [6] and uses acronyms to refer to these techniques. Table 2 presents the components of the architecture covered by existing testing techniques. MBT stands for Model-Based Techniques, C/RT for Capture/Replay Techniques and rand for Random techniques.

Table 2. Components of the architecture covered by testing techniques [P = Partial coverage, NC = No Coverage].

Techniques	Input devices type	Input chain manager	Global interaction technique	Dialogue	Rendering system	Output chain manager	Output devices type
MBT e.g. [6]	NC	NC	NC	P	NC	P	NC
C/RT e.g. [6]	NC	NC	NC	P	NC	P	NC
Rand e.g. [6]	NC	NC	NC	P	NC	P	NC
[10, 26]	NC	P	NC	P	NC	P	NC
[38]	NC	P	NC	P	NC	P	NC

MBT approaches are mainly used to test the behavior of the **Dialogue** component of the AUT. However, [26] proposes, in addition, to use model-based descriptions of multimodal interaction techniques for testing. In MIODMIT terms it means that [26] supports testing of part of the **Input Chains Manager** component. [10] discusses the testing of multimodal interactive system, taking into account the **Input Chains Manager**.

On modern operating systems (e.g. Android), the permission mechanism allows the user to restrict application access to input and output devices, affecting the **Input/Output Chain Manager**. By developing Permission-Aware GUI Testing on Android, [37] supports partial testing of this function handled by the **Input/Output Chain Manager**.

Three columns in Table 2 are not covered by any previous work. Another concern is that existing techniques only support partially the testing of the covered components. Indeed, the **Dialogue** is mostly tested through the state of GUI widgets. The **Output Chain Manager** is mainly tested by checking properties of the GUI widgets via their public accessors, so their rendering is not assessed. On this aspect, we note that the emergence of techniques based on computer vision (in order to assess what the users will be seeing), such as [18] will be of great help to support automated testing.

Overall, we note that there is a need for new dedicated testing techniques to cover all the elements of the architecture of interactive systems.

7 Conclusion

This paper has presented an architecture-driven approach to support testing of interactive systems. This Approach exploits MIODMIT architecture that has been used in multiple domains such as interactive cockpits of large civil aircrafts of multimodal interfaces for new cockpits of helicopters as well as desktop interactive applications [9]. This paper has presented numerous specificities of testing of interactive systems with respect to "standard" software testing. We have shown that known problems in testing interactive systems can be positioned on one or multiple elements of MIODMIT providing details on unit and integration tests problems for interactive systems.

One of the key elements of MIODMIT is its genericity and its capability of handling multiple input and output devices. This is critical for interactive systems engineering as new devices and new interaction techniques are frequently proposed to increase the bandwidth between operators and computing systems. For instance, MIODMIT handles devices such as Kinect, Leap (Motion), speech recognition systems, multiple parallel graphical input devices, just to name a few [14] but was not presented due to space constraints.

In this paper we have also presented how user testing (or more generally user studies) connects to the interactive systems testing which is the focus of this paper. The H-MIODMIT architecture highlights the fact that interactive systems are meant to be used by users and that this specific component (the user as a human) may add (but also relax) constraints on interactive systems testing. Beyond, if user studies needs are known and described while developing interactive systems, software specifications and software testing techniques can support those activities as demonstrated in [33].

Future work will be dedicated to the definition of techniques to support unit testing of each component of MIODMIT but also integration tests (e.g. the immediate feedback loop presented in the paper). The objective is to provide interactive systems developers with adequate solutions in order to test their application beyond the classical "test coverage" and "non regression testing" measures that are unfortunately meaningless when interactive systems are considered.

References

1. Accot, J., Chatty, S., Palanque, P.: A formal description of low level interaction and its application to multimodal interactive systems. In: Bodart, F., Vanderdonckt, J. (eds.) DSV-IS 1996. Eurographics, pp. 92–104. Springer, Vienna (1996). https://doi.org/10.1007/978-3-7091-7491-3_5
2. Accot, J., Chatty, S., Maury, S., Palanque, P.: Formal transducers: models of devices and building bricks for the design of highly interactive systems. In: Harrison, M.D., Torres, J.C. (eds.) DSV-IS 1997. Eurographics, pp. 143–159. Springer, Vienna (1997). https://doi.org/10.1007/978-3-7091-6878-3_10

3. Abbaspour Asadollah, S., Inam, R., Hansson, H.: A survey on testing for cyber physical system. In: El-Fakih, K., Barlas, G., Yevtushenko, N. (eds.) ICTSS 2015. LNCS, vol. 9447, pp. 194–207. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25945-1_12
4. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**, 11–33 (2004)
5. Bailly, G., Lecolinet, E., Nigay, L.: Visual menu techniques. ACM Comput. Surv. **49**(4), 60:1–60:41 (2017)
6. Banerjee, I., Nguyen, B., Garousi, V., Memon, A.M.: Graphical user interface (GUI) testing: systematic mapping and repository. Inf. Softw. Technol. **55**, 1679–1694 (2013)
7. Bass, L., et al.: The arch model: Seeheim revisited. In: User Interface Developpers' Workshop (1991)
8. Bellik, Y.: Multimodal interfaces: concepts, models and architecture, Ph.D. thesis, University Paris-South 11, Orsay (1995). (in French)
9. Bernhaupt, R., Cronel, M., Manciet, F., Martinie, C., Palanque, P.: Transparent automation for assessing and designing better interactions between operators and partly-autonomous interactive systems. In: ATACCS 2015, pp. 129–139 (2015)
10. Bouchet, J., Madani, L., Nigay, L., Oriat, C., Parissis, I.: Formal testing of multimodal interactive systems. In: Gulliksen, J., Harning, M.B., Palanque, P., van der Veer, G.C., Wesson, J. (eds.) EIS 2007. LNCS, vol. 4940, pp. 36–52. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92698-6_3
11. Bourque, P., Fairley, R.E., IEEE Computer Society: Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0. IEEE Computer Society Press, Los Alamitos (2014)
12. Campos, J.C., et al.: A more intelligent test case generation approach through task models manipulation. In: Proceedings of the ACM HCI. EICS, vol. 1, pp. 9:1–9:20 (2017)
13. Cockton, G., Woolrych, A.: Understanding inspection methods: lessons from an assessment of heuristic evaluation. In: Blandford, A., Vanderdonckt, J., Gray, P. (eds.) People and Computers XV—Interaction without Frontiers, pp. 171–191. Springer, London (2001). https://doi.org/10.1007/978-1-4471-0353-0_11
14. Cronel, M., Dumas, B., Palanque, P., Canny, A.: MIODMIT: a generic architecture for dynamic multimodal interactive systems. In: Bogdan, C., et al. (eds.) Human-Centered and Error-Resilient Systems Development, HCSE 2018. LNCS, vol. 11262, pp. 109–129. Springer, Cham (2018)
15. Cuenca, F., Coninx, K., Vanacken, D., Luyten, K.: Graphical toolkits for rapid prototyping of multimodal systems: a survey. Interact. Comput. **27**, 470–488 (2015)
16. Dragicevic, P., Fekete, J.D.: Input device selection and interaction configuration with ICON. In: Blandford, A., Vanderdonckt, J., Gray, P. (eds.) People and Computers XV—Interaction without Frontiers, pp. 543–558. Springer, London (2001). https://doi.org/10.1007/978-1-4471-0353-0_34
17. Dragicevic, P.: Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables. Ph.D. Université de Nantes (2004). (in French)
18. Göransson, B., Gulliksen, J., Boivie, I.: The usability design process - integrating user-centered systems design in the software development process. Softw. Process Improv. Pract. **8**(2), 111–131 (2003)
19. Greenberg, S.: Working through task-centered system design. In: Diaper, D., Stanton, N. (eds.) The Handbook of Task Analysis for Human-Computer Interaction. Lawrence Erlbaum Associates (2002)
20. Ha, T.T., Ghaffari, R.: Simulating Single and Multi-Touch Events for Testing a Touch Panel (2012). https://patents.google.com/patent/US20120280934A1/en

21. ISO 9241-11. Ergonomics of human system interaction - Part 11. Usability: Definitions and concepts (2018)
22. Kelley, J.F.: An iterative design methodology for user-friendly natural language office information applications. ACM Trans. Inf. Syst. **2**(1), 26–41 (1984)
23. Lalanne, D., Nigay, L., Palanque, P., Robinson, P., Vanderdonckt, J., Ladry, J.F.: Fusion engines for multimodal input: a survey. In: ICMI, pp. 153–160. ACM (2009)
24. Lee, J.S., et al.: A 0.4 V driving multi-touch capacitive sensor with the driving signal frequency set to (n + 0.5) times the inverse of the LCD VCOM noise period. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 682–685 (2014)
25. Lelli, V., Blouin, A., Baudry, B.: Classifying and qualifying GUI defects. Presented at the 8th IEEE International Conference on Software Testing, Verification and Validation, 13 April 2015
26. Lelli, V., Blouin, A., Baudry, B., Coulon, F.: On model-based testing advanced GUIs. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10 (2015)
27. Memon, A.M., Soffa, M.L., Pollack, M.E.: Coverage criteria for GUI testing. In: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 256–267. ACM, New York (2001)
28. Memon, A.M.: GUI testing: pitfalls and process. Computer **35**(8), 87–88 (2002)
29. Memon, A.M.: A comprehensive framework for testing graphical user interfaces. Ph.D. thesis, University of Pittsburgh, Pittsburgh (2001)
30. Memon, A.M., Nguyen, B.N.: Advances in automated model-based system testing of software applications with a GUI front-end. In: Zelkowitz, M.V. (ed.) Advances in Computers, pp. 121–162. Elsevier (2010)
31. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.: GUITAR: an innovative tool for automated testing of GUI-driven software. Autom. Softw. Eng. **21**, 65–105 (2014)
32. Nielsen, J.: Usability Engineering. Morgan Kaufmann, San Francisco (1994)
33. Palanque, P., Barboni, E., Martinie, C., Navarre, D., Winckler, M.: A model-based approach for supporting engineering usability evaluation of interaction techniques. In: Proceedings of EICS 2011, pp. 21–30. ACM (2011)
34. Pfaff, G.E. (ed.): Proceedings of IFIP/EG Workshop on User Interface Management Systems (November 1983, Seeheim, FRG). Springer, Berlin (1985)
35. Pirker, M., Bernhaupt, R.: Measuring user experience in the living room: results from an ethnographically oriented field study indicating major evaluation factors. In: Proceedings of the 9th European Conference on Interactive TV and Video (EuroITV 2011), pp. 79–82. ACM, New York (2011)
36. Rowley, D.E.: Usability testing in the field: bringing the laboratory to the user. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 1994), pp. 252–257. ACM, New York (1994)
37. Sadeghi, A., Jabbarvand, R., Malek, S.: PATDroid: permission-aware GUI testing of android. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 220–232. ACM, New York (2017)
38. Song, W., Qian, X., Huang, J.: EHBDroid: beyond GUI testing for android applications. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 27–37. IEEE Press, Piscataway (2017)
39. Thimbleby, H.: Reasons to question seven segment displays. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1431–1440. ACM, New York (2013)
40. Vu, T., et al.: Distinguishing users with capacitive touch communication. In: Mobicom 2012, pp. 197–208 (2012)