Model-Based Testing of GUI Applications Featuring Dynamic Instanciation of Widgets

Alexandre Canny
ICS-IRIT
Université Toulouse III Paul Sabatier
Toulouse, France
alexandre.canny@irit.fr

Philippe Palanque
ICS-IRIT
Université Toulouse III Paul Sabatier
Toulouse, France
palanque@irit.fr

David Navarre
ICS-IRIT
Université Toulouse III Paul Sabatier
Toulouse, France
navarre@irit.fr

***Abstract*— The testing of applications with a Graphical User Interface (GUI) is a complex activity because of the infinity of possible event sequences. In the field of GUI Testing, model-based approaches based on reverse engineering of GUI application have been proposed to generate test cases. Unfortunately, evidences show that these techniques do not support some of the features of modern GUI applications. These features include dynamic widgets instantiation or advanced interaction techniques (e.g. multi-touch). In this paper, we propose to build models of the applications from requirements, as it is standard practice in Model-Based Testing. To do so, we identified ICO (Interactive Cooperative Object) as one of the modelling techniques allowing the description of complex GUI behavior. We demonstrate that this notation is suitable for generating test cases targeting complex GUI applications in a process derived from the standard Model-Based Testing process.**

Keywords—GUI Testing, Model-Based Testing, User Interface Description Languages

I. INTRODUCTION

The testing of applications with a Graphical-User Interface is known to be a complex activity [30], especially due to the unpredictability of the human behavior as well as to the infinite number of possible event sequences [9]. To cope with these challenges, model-based testing techniques have been developed to try to generate relevant test sequences without relying on manual scripting or capture and replay of tester's interactions. These model-based techniques mostly relies on the generation of test cases from models built dynamically while exploring the execution space of the Application Under Test (AUT) [37] (i.e. models built by reverse engineering). Unfortunately, evidences that these approaches tend to be less effective than less structured ones with a significant random aspect (i.e. random and systematic techniques [37]) has been found for both mobile [13] and desktop applications [37]. Furthermore, evidences that these approaches does not enable the testing of increasingly complex GUI applications have also been provided, for instance because they do not support ad-hoc and non-standard widgets [25] or advanced interaction techniques [10][25] (e.g. multi-touch ones).

If in Model-Based approaches for GUI Testing models are built dynamically while exploring the execution space of the AUT, the process of Model-Based Testing as defined by Utting et al. [38] proposes to build the models from the requirements of the application. One of the main challenge in Model-Based Testing is that it requires modelling techniques that are expressive enough to describe the behavior of the AUT. Increasingly complex GUI applications may features multimodal interaction techniques (e.g. "put-that-there" voice+gesture interaction [7]), dynamic instantiation of input devices (e.g. adding fingers on a touch-screen), scaling, dynamic instantiation of GUI widgets, etc. Each of these features raises challenges for both modelling and testing and this paper focuses on the challenges related with MBT of applications featuring dynamic instantiation of GUI widgets. To overcome these challenges, we investigated the expressiveness of several User Interface Description Languages (UDILs) to establish which ones are expressive enough to allow the generation of test cases for such applications. We identified ICO (Interactive Cooperative Object) as the best fit for this purpose. We illustrate the possibilities offered by this notation by modelling an Air Traffic Control Application. Additionally, we propose a process derived from [38] to enable both the implementation and the testing of a GUI application from the ICO models. This process relies on random-constrained simulation of the models to build relevant test cases while allowing to cope with the potentially infinite number of event sequences that can be generated.

This paper is structured as follow: section II presents the challenges in GUI testing and the current state of the art in modelling of GUI applications. Section III presents the ATC application we use as a case study and section IV presents ICO, the notation we identified as the most expressive for describing GUI applications featuring dynamic instantiation of GUI widgets. Section V demonstrates the ability to model the behavior and the presentation of the ATC application with ICO and section VI presents our process for building and testing the AUT from ICO models. Ultimately, section VII concludes this paper in addition to discussing future work.

II. CHALLENGES AND CURRENT STATE OF THE ART IN TESTING AND MODELLING OF GUI APPLICATIONS

This paper aims at providing solutions to challenges encountered in GUI Testing by generating test cases from models built prior to development. As the understanding of these problem is a key in building new test case generation techniques, this section first present some of the challenge

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00029

encountered in GUI testing before discussing how expressive a description technique must be to enable test cases generation from models built prior to development.

A. Challenges and Current State of the Art in GUI Testing

GUI Testing is defined by Banerjee et al. as the testing of an application that has a Graphical User Interface solely by performing sequence of events (e.g. click on button, enter text) on GUI widgets (e.g. buttons, text-fields) [2]. The key issue here is that the space of all possible event sequences that may be executed is extremely large, in principle infinite [2]. Thus test case generation is made difficult and manually generated test scripts usually offers little coverage of the event sequences [2] as it would be too long and costly for testers to comprehensively explore the execution space of the AUT. These manual test scripts are usually produced using scripting language (e.g. Abbot [1], Selenium Web Driver [2]) or Capture/Replay techniques (where user interaction are logged for being replayed latter in regression testing, (e.g. GUICat [12]; Test Automation FX[3])).

Automatic GUI Testing techniques aims at solving some of the issues brought by manual GUI testing approaches by offering better coverage of the event sequences through automated exploration of the execution space. Various techniques enables automatic GUI testing, such as [37]:

- **Random-Walk** (e.g. Testar-Random [40], UI/Application Exerciser Monkey[4]) which randomly plays events on available GUI widgets and is particularly relevant for crash-testing (making sure that no event sequence lead to the AUT stop responding/crashes);
- **Model-Based Techniques** (e.g. ABT [26], GUITAR [30], Testar-QLEARNING [40]) that drive the generation of test cases with a model of the GUI of the AUT. Model-based techniques use graph models built dynamically while exploring the execution space of the AUT [37].

The later approach is rather original as Model-Based Testing techniques usually relies on models built from the requirements specified for the AUT [38]. This may explain why, even though Model-Based Testing is often praised for its effectiveness, GUI Testing using Model-Based techniques is less effective than random techniques [13][37]. Unfortunately, most of the existing GUI applications were not built from models. Thus, these techniques makes sense, especially if we consider that building models allows to reason on the application behavior.

Yet, while models of applications still are not the norm, the User-Centered Design process in Human-Computer Interaction (HCI) involves the production of models of the user interactions with the applications, called Task-Models. These models are more likely to exist for modern applications, and techniques has been proposed for building test cases from them [8]. Beyond HCI, current trends towards pushing Model-Based Software Engineering (MBSE) in Software Engineering [16] makes it conceivable that in a foreseeable future, more and more GUI applications will be built from some sort of model, but which? Advanced GUI applications features behavior that cannot be described [25][10] using the modelling techniques found in GUI Testing [27]. These behaviors includes ones based on non-standard[5] GUI widgets (e.g. drawing area of a drawing tool), dynamic instantiation of GUI widgets or actions tied to multiple events sources (e.g. multiple fingers during multi-touch interaction). The following sub-section discusses the state of the art in describing GUI applications to help in identifying modelling techniques that support the description of these behaviors to enable test cases generation, with focus on dynamic instantiation of GUI widgets.

B. Modelling of GUI Applications

Model-Based Testing (MBT) of software relies on explicit behavior models of a system to derive test cases [38]. The complexity of deriving comprehensive test cases increases with the complexity of the AUT: with complex behaviors come the need for modelling techniques expressive enough to describe the complexity of the AUT behavior [10]. Any behavior that cannot be described cannot have test cases generated for using MBT. The techniques for describing the behavior of GUI applications are called User Interface Description Languages (UIDL). In [17], Hamon et al. identified several characteristics UIDLs must handle to enable the comprehensive description of user interfaces behaviors. These characteristics include some common to all GUI applications while other applies specifically to applications featuring dynamic instantiation of widgets, multimodal interaction, etc. As the focus of this paper is on the testing of applications featuring dynamic instantiation of GUI widgets, we investigated the following subset of the characteristics from [17]:

- **Data description** for describing the objects and values so information such as widget location at a given time is know;
- **State/Event representation** as commonly described is Model-Based techniques for GUI Testing [27];
- **Qualitative time** between two consecutive model elements aims at representing ordering of actions such as precedence, succession, and simultaneity;
- **Quantitative time** between two consecutive model elements represents behavioral temporal evolutions related to a given amount of time (usually expressed in milliseconds);
- **Concurrent behavior** representation is necessary when the interactive systems feature multimodal interactions;
- **Dynamic instantiation** of widgets is a characteristic required for the description of interfaces where objects are not available at the creation of the interface as, for

[1] http://abbot.sourceforge.net/doc/overview.shtml
[2] https://selenium.dev/documentation/en/webdriver/
[3] http://www.testautomationfx.com/
[4] https://developer.android.com/studio/test/monkey

[5] We call non-standard the GUI widgets that were built purposely for an application and are not found in the GUI widgets library of programming languages.

instance, in desktop-like interfaces where new icons are created according to user actions.

Table 1 details the expressiveness of UIDLs regarding these characteristics. For all characteristics, there are three possible values:

- **Yes** means that that characteristic is explicitly handled by the UIDL;
- **No** means that the characteristic is not explicitly handled;
- **Code** means that the characteristic is made explicit but only at the code level and is thus not a construct of the UIDL.

Table 1. UIDL expressiveness for the main characteristics of modern application (adapted from [17]).

		Constraint		Code Based			Flow Based			State Based			Petri Nets		
		ConstraintUS [31]	Squeak [11]	XISL [24]	GeForMT [33]	Continued in [17]	Marigold [39]	ICON [14]	Continued in [17]	Swinstate [1]	HephaisTK [15]	Continued in [17]	Hinckley [20]	ICO [18]	CPN [22]
Data Description						...			...			...			
State Representation						...			...			...			
Event Representation						...			...			...			
Time	Qualitative between two consecutive model elements					...			...			...			
	Quantitative between two consecutive model elements					...			...			...			
	Quantitative over non consecutive elements					...			...			...			
Concurrent Behavior						...			...			...			
Dynamic Instanciation of Widgets						...			...			...			

Concepts covered in this paper — Yes — Code — No

Due to space constraint, we won't detail on the handling of all these characteristics by all the UIDLs we present. Yet, thanks to an analysis based on categories of UIDLs, we can observe that state-based and flow-based approach (also used in MBT of GUI applications, as shown in [27]) are not suitable for an effective description of advanced GUI applications. Additionally, we observe that UIDLs based on Petri Nets augmented with temporal aspects (time was intentionally avoided in Petri's original work [36]) such as CPN [22] and ICO [18] are amongst the most expressive UIDLs. Yet, only ICO [18] appears to be expressive enough to support the characteristics required for comprehensively describing GUI application featuring dynamic instantiation of GUI widgets. In the following of this paper, we will introduce the ICO UIDL and demonstrate that its expressiveness combined with tool support enables test cases generation for such applications on an example based on an Air-Traffic Control Application.

III. Example of an Application Featuring Dynamic Instanciation of GUI Widgets: The Air Traffic Control System

The case study in this paper focuses on the GUI aspects of an Air Traffic Control application. Nowadays, the airspace is divided in sectors, each of them being controlled by two air traffic controllers managing different tasks and working in a cooperative way. The air traffic controller has at his disposal a workstation for handling the traffic over a given sector, by communicating with the pilots in the planes currently flying in the sector. This section presents an overview of the ATC application and the requirements it must fulfills.

A. Overview of the ATC Application

From his/her position, an air traffic controller can interact with aircraft pilots using two means: voice and datalink messages. For the Air Traffic Controller, the ATC Application (see Fig. 1) is the entry point for every exchange s/he wants to initiate with the pilot. The air traffic controllers may request the pilot to contact them using the radio (**VOICE** command) or send them command (text message through datalink) to change the course of their flights. These commands are the following and are all followed by a parameter value:

- **FREQ** for asking the pilot to switch from one radio frequency to another one;
- **CFL** for notifying the new Cleared Flight Level for the flight (i.e. the altitude it is allowed to fly at);
- **SPEED** for requesting a speed change;
- **HEAD** for requesting an heading change;
- **BEACON** for requesting a change of the route of the place by designating the next beacon it has to fly over.

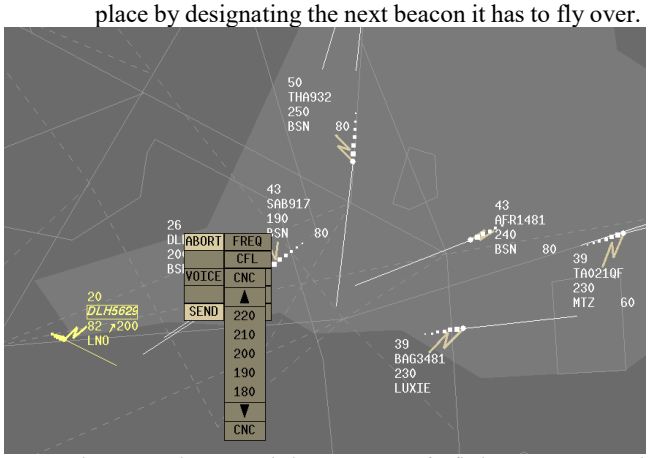

Fig. 1. The ATC Application with the pop-up menu for flight DLH5629 opened and SPEED sub-menu selected.

The GUI of the ATC application (Fig. 1) can be split in three:

- the light grey part which correspond to the sector handled by the controller. In this sector, white line represent plane routes. On these routes planes are represented by a succession of white dots, the first one being the actual position of the plane and the smaller ones its previous positions. Along with these dots information concerning the flight are displayed: the call sign of the flight, its speed, its heading and the next beacon it is supposed to fly over;
- the dark grey part which represent all the outside of the controlled sector;
- the open menu used by the controller for entering information. This menu is a pop-up menu that appears when the air traffic controller clicks on the label of a flight. When it pops-up, the menu is split in two parts:

- On the left side the direct command menu (no parameter is needed) which offers three commands, **SEND** and **ABORT** (for sending or cancelling the current data-link command) and **VOICE** for asking the pilots to call the controller using the radio (sending the request is still required once VOICE is selected);
- On the right side of the menu the user can select one of five commands which need parameter (**FREQ**, **CFL**, **SPEED**, **HEAD** and **BEACON**). Each time the controller selects one of these commands, another pop-up menu appears that allows for entering the parameter of the command. By pressing CNC (cancel) this pop-up menu is closed, by selecting a value, the parameter is set. The air traffic controller can choose amongst additional values by scrolling amongst the available ones using the arrows at the top and bottom of the value list.

B. Requirements for the ATC Application

The following set of requirements have to be fulfilled by the ATC application:

- A control order is only received by one plane;
- Any request sent by a controller will be received at some time by a pilot;
- A control order is sent to only one plane (only one plane can be selected at a time);
- For each control order, only one information is sent;
- All control orders finish by either Abort or Send;
- It is not possible to build several orders at a time.

These requirements can be classified in two categories: high-level requirements related to the very semantics of air-traffic control, and lower level ones, related to interaction techniques. In the following of this paper, we will discuss on how to model and test some of these requirements on GUI widgets representing the aircrafts that are the dynamically instantiated widgets in this application.

IV. The ICO User Interface Description Language

By analyzing the expressiveness of UIDLs, we have established that ICO (Interactive Cooperative Objet) is one of the most relevant for describing the characteristics of advanced GUI applications. In this section, we introduce ICO and details how it enables the description of the behavior of the GUI.

A. Informal Presentation of ICO

ICOs (Interactive Cooperatives Objects) are a formal description technique dedicated to the specification of interactive systems. ICO uses concepts borrowed from the object-oriented approach (dynamic instantiation, classification, encapsulation, inheritance, client/server relationship) to describe the structural or static aspects of systems, and uses high-level Petri nets to describe their dynamics or behavior. The ICO notation is based on a behavioral description of the interactive system using the Cooperative objects formalism that describes how the object reacts to external stimuli according to its inner state. This behavior, called the Object Control Structure (ObCS) is described by means of Object Petri Net (OPN). An ObCS can have multiple places and transitions that are linked with arcs as with standard Petri nets. As an extension to these standard arcs, ICO allows using test arcs and inhibitor arcs. Each place has an initial marking (represented by one or several tokens in the place) describing the initial state of the system. To convey the relationship between the evolution of the behavioral model and its impact on the GUI, ICO uses rendering and activation functions, tying places and transitions to properties/event handlers of the GUI widgets. The ICO notation is fully supported by a CASE tool called PetShop [4][35]. ICO has already been applied in the field of Air Traffic Control interactive applications [28], space command and control ground systems [34], interactive military [5] or civil cockpits [3]. In previous work, the notation capability to support the testing of multi-touch interaction technique has been demonstrated [10] (yet, as mentioned previously, testing of interaction techniques comes with its own set of challenges that are out of scope of this paper).

B. Notation for the description of the behavioral part of the GUI

The ICO notation uses concepts borrowed from Petri nets. Fig. 4 presents an example of such Petri net, which is one of the ICO model we use to describe the ATC application. It is made of **places** (oval shapes, e.g. Flights), **transitions** (rectangular **shapes**, e.g. addingFlight) and arcs. The **marking** of this Petri net is the distribution of **tokens** (numbered circles in the places) in the different places at a given time (Fig. 4.a and Fig. 4.b presents two different markings). As ICO relies on Object Petri nets, tokens can carry values, from generic types (e.g. int, float) to complex objects, including instances of other ObCSes. Fig. 2 details the marking of the place "Flight" found in Fig. 4.a. Each line of the table in Fig. 2 describes tokens. Here, there are 3 lines with a multiplicity of 1, meaning that there are three token in the place each carrying a different object. In that particular case, the objects carried are three instances of ObCSes corresponding to the ICO description of SimpleFlights.

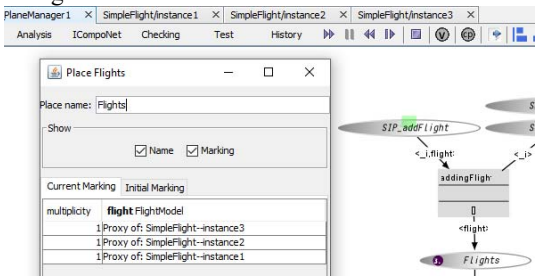

Fig. 2. Marking of the place "Flights" of the model PlaneManager1.

The marking of an ICO can change after **invocations** or **firing of armed transitions**. There are two kinds of transitions in ICO: **standard transitions** and **event transitions**. They are both armed whenever there are tokens matching the **pre-conditions of the transition** in the places from which there are

incoming arcs[6]. For instance, in Fig. 3, the "transition1" can be fired if "place0" contains a token carrying the value 5 (initially, it contains 2 of them). Note that only one token is consumed when the transition is fired, so 2 step are required to consumer the two tokens with value==5. Note that, in Petshop, armed transition are represented in purple, non-armed ones in grey (see Fig. 3).

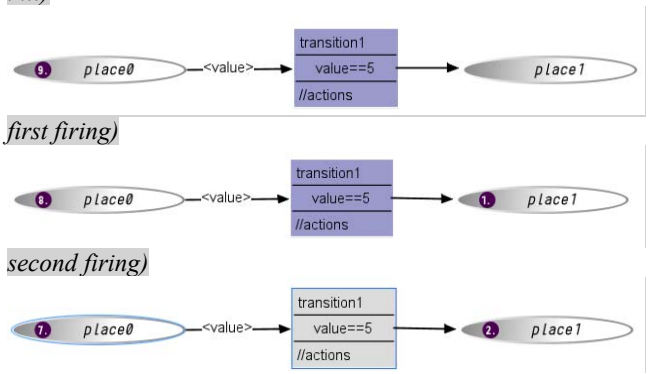

Fig. 3. Example of a transition with a pre-condition. Initially, place0 contains 9 tokens, 2 of which carrying a value of 5.

Event transitions differ from standard ones as they are fired if and only if they are armed and the event they are listening for has been produced. **Event transitions are the ones relevant for generating event sequences for GUI Testing.** In the bottom-left of Fig. 4.b, the transition "sendingCommand" is an example of event transition. Next to its title, "::send" indicates the name of the event the transition is listening for. Below details are provided regarding this event such as the expected source ("from" *menu*, another model describing a menu), the parameters of the event (if any, in that case the *command* selected in the menu) and the preconditions on the event. At the very bottom of the transitions are **invocations** on other ObCSes. **Invocations** enables unicast and synchronous communication, represented by method calls in ICO. When an ObCS offers invocations, they are each mapped into a set of three places representing three communication ports (the invocation input, output and exception ports). For instance, in Fig. 4.a, the places SIP_addFlight, SOP_addFlight and SEP_addFlight are the input, output and exception ports of the method addFlight. When this method is called, a token is created, holding the parameters of the invocation and is put in place SIP_addFlight. ObCSes can invoke each other: for instance, in the event transition "sendingCommand", the bottom part contains invocations on both the ObCS describing the menu (menu.close()) and the flight (flight.sendCommand(command)).

C. Notation for description of the presentation part of the GUI (rendering and activation)

In addition to the Petri net describing the behavior of the GUI, ICO description defines the relationship between the behavioral model and the GUI elements. To do so, it makes use of **rendering** and **activation functions**. Rendering (how the changes in the behavioral model affects the GUI) and activation (how the actions on the GUI widgets affects the behavioral model) functions are provided in the form of tables such as Table 2 and Table 3, respectively.

The **rendering function** (see Table 2) associates the changes of the places markings with rendering methods on GUI elements. For example, in Table 2, rendering methods for the place "Flights" of Fig. 4 are associated to the following event: token entering (token_enter) and leaving (token_removed) the place as well as resetting the marking (marking_reset) of the place (i.e. going back to initial marking). Table 2 shows that

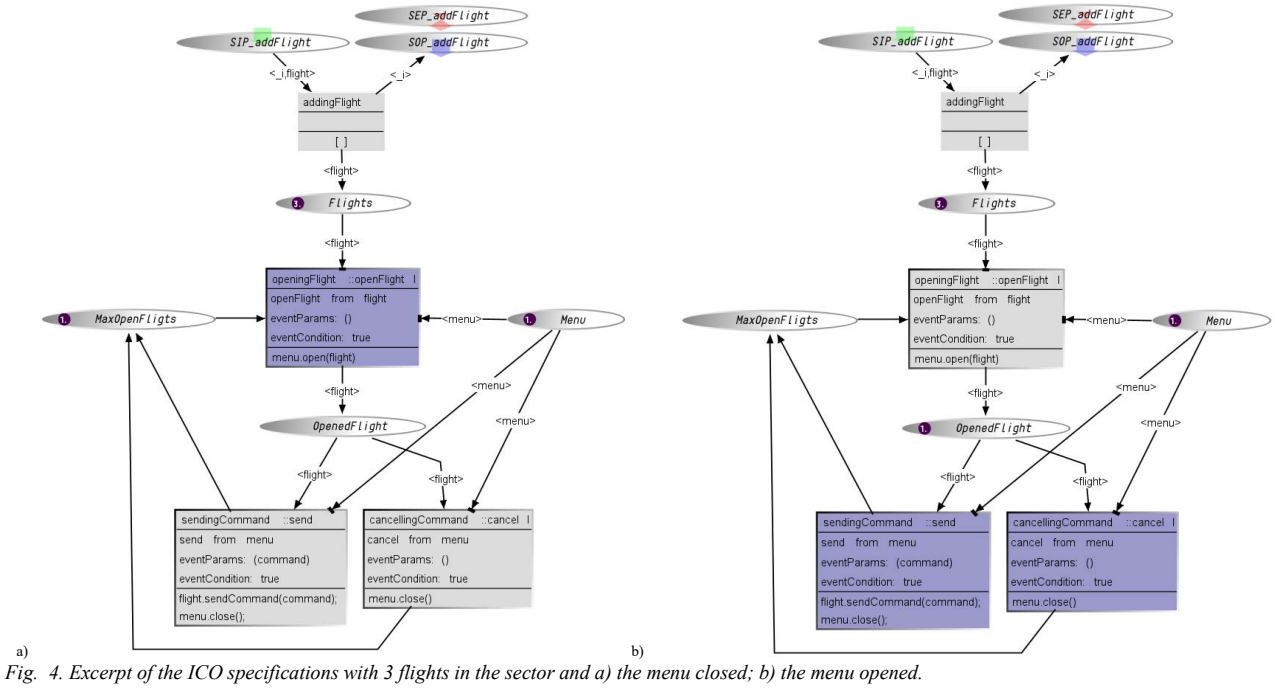

Fig. 4. Excerpt of the ICO specifications with 3 flights in the sector and a) the menu closed; b) the menu opened.

whenever a token enters the place "Flights", a component (widget) is added to the drawing area of the radar screen. Similarly, a widget is removed whenever a token leaves the place.

Table 2. Excerpt of the rendering function associated with Fig. 4.

Place	ObCS Event	Rendering method
Flights	token_enter	add(component) of the radar drawing area
Flights	token_removed	remove(component) of the radar drawing area
Flight	marking_reset	removeAll() of the radar drawing area

The **activation function** (see Table 3) associates the GUI elements with the **event transition**s bi-directionally. On one hand, the user events (e.g. mouseClicked on a flight) are associated with the event of a transition (e.g. openFlight). On the other, the states of event transitions (armed or not) are associated with activation methods (e.g. setEnabled(bool)). Table 3 shows that the "openFlight" event in the "openingFlight" transition from Fig. 4 is associated with a mouseClicked event on a flight. Table 3 also indicates that whenever the associated transition is armed, flights must be enabled (clickable), disabled (non-clickable) otherwise. Note that even though they are not visible when no flights are open, SEND and ABORT menu item are enabled/disabled according to the state of their associated event transitions, thus preventing the use of accelerator keys, if any.

Table 3. Excerpt of the activation function associated with Fig. 4.

User Event	ObCS Event	Activation method
mouseClicked on a flight	openFlight	setEnabled(bool) of flights
actionPerformed on SEND	send	setEnabled(bool) of SEND menu item
actionPerformed on ABORT	cancel	setEnabled(bool) of ABORT menu item

V. Modelling of the ATC Application GUI Using ICO

The modelling of the ATC application using ICO requires the description (using ObCS and rendering/activation functions) of three elements 1) the radar image, 2) flight and 3) menu. This section introduces the description of these three elements and highlights their means of supporting dynamic instantiation as well as the characteristics that enables them to support the requirement previously listed.

A. Description of the Radar Image (Main Screen)

The "main screen" of the ATC application consists in an image of the sector controlled by the air traffic controller on which aircraft are drawn after being dynamically instantiated. Fig. 4 presents two markings of the model describing the radar image behavior.

The dynamic instantiation of flights is supported through the addFlight invocation. The "addingFlight" transition consumes the token placed in the "SIP_addFlight" and add them to the Flights place.

Interaction with the dynamically instantiated aircraft must be controlled as it is required that:

- A control order is sent to only one plane (only one plane can be selected at a time);
- All control orders finish by either Abort or Send;

With regards to GUI widgets, this means that:

- (req1) A flight should be clickable if and only if there is no flight already opened;
- (req2) Closing of the pop-up menu is allowed only as a result of a click on Send or Abort (as opposed to the behavior of contextual menus in standard applications, i.e. ones that are closing as soon as another is opened).

In Fig. 4.a, three flights are on the sector (place "Flights") but no flights are opened (place "OpenedFlights"). Thus, it is possible to "open" a flight (the event transition "OpeningFlight is armed). When the "openFlight" event is raised by a flight, the marking of the ICO changes to the one presented in Fig. 4.b. The firing of the transition led to the consumption of the token in the place "MaxOpenFlight" (the flight token is copied as a test arc is used). This disarmed the transition "openingFlight". At this point, and according to the activation function (Table 3), all the flights become non-clickable, fulfilling (req1).

In Fig. 4.b, a flight is opened, which led to the opening of the menu (action "menu.open(flight)" in the "openingFlight" transition). The fulfillment of (req2) is ensured by the fact that the "menu.close()" invocation is performed only on the "sendingCommand" and "cancellingCommand" transitions. These transitions respectively handle "send" and "cancel" events from the menu. When one of these transitions is fired, the token in "OpenedFlight" is consumed, and a token is added to MaxOpenFlights: the model returns to the marking of Fig. 4.a.

B. Description of a Flight

The modelling of the behavior of a flight enables the verification of properties that goes beyond the requirements listed in the presentation in the case study. For the ATC Application to be usable, the data it provides must be accurate. Modelling the behavior of aircrafts enables the verification of widgets properties such as location, label for flight number, speed, etc. ICO allows these parameters to be provided at every instantiation of a model and one may allow to update them at any time via **event transitions** or **invocation** once the model is instantiated. Due to space constraint, we do not present the model of a SimpleFlight, which is mainly composed of getters and setters (in the form of invocation) for properties such as latitude, longitude or speed and of getters for properties such as position (derived from latitude and longitude).

C. Description of the Menu

By design, the menu must prevent sending inconsistent data to the aircraft. For instance, "VOICE 12500" is an invalid command as VOICE is non-parameterized. Similarly, the command "HEAD" misses the heading parameter value. To prevent the production of such command explicitly, the model of the menu contains a pattern similar to the one presented in Fig. 5. This pattern forces the deactivation of the SEND button (see the two event transitions at the bottom of Fig. 5) until either the "VOICE" command is selected (armed transition :

"commandWithoutParam") or a command that is not the "VOICE" one is selected with a parameter set (place "PARAMETER") .

Note that to prevent sending a parameter incompatible with the selected command, this pattern is designed to remove any token from the place "PARAMETER" when a click on a new command is done (a token is added in "CLEAR PARAMETER"; if a parameter was set, the token is consumed by the transition "clearingParameter". The arc between the place "PARAMETER" and the "noParameterToClear" transition is an inhibitor one: it implies that the transition is armed only if "PARAMETER" is empty).

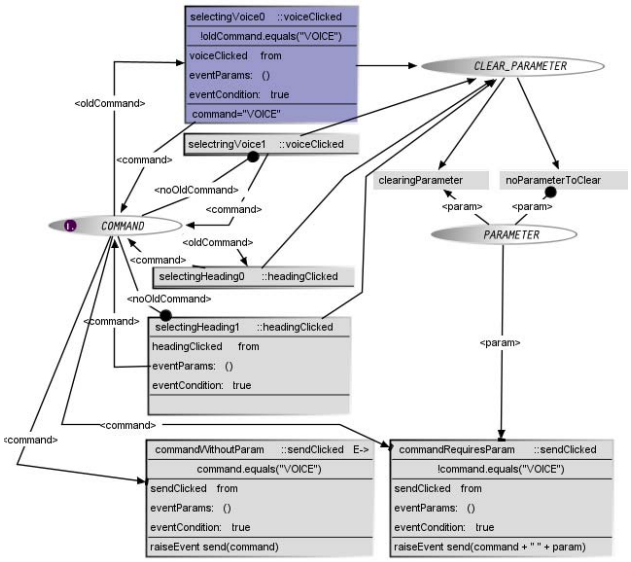

Fig. 5. Example of a pattern preventing clicking on the "send" button with an incorrect command.

VI. A Process For Model-Based-Testing of Applications Featuring Dynamic Instanciation of GUI Widgets

So far, we showed that ICO enables the description of GUI applications featuring dynamic instantiation of GUI and carries some properties regarding GUI widgets thanks to the rendering and activations functions. In this section, we present a process inspired by [38] to take advantage of the ICO expressiveness to enable:

- The implementation of the AUT using ICO as specification;
- The generation of test cases for the AUT from the ICO models.

To cope with the challenges associated with test cases generation, this process involves a random exploration of the application model, constrained by test selection criteria. This allow selecting relevant sample of the infinite execution space of applications during generation. Fig. 6 presents our process that uses the requirements for the AUT as inputs (top of Fig. 6) and relies on the exact same models for driving the implementation and the testing activities. The following sections detail the activities of this process and discusses the challenges they raises, especially regarding the generation of relevant invocations and event parameters.

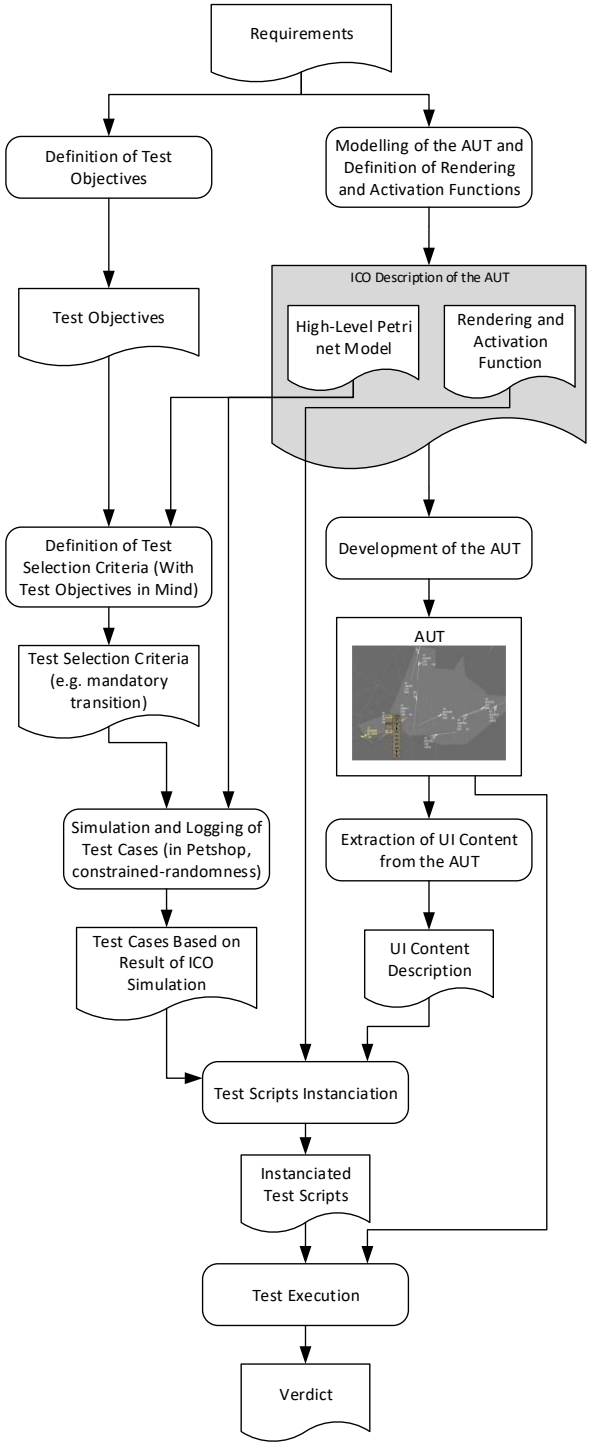

Fig. 6. The Process of Random-Model Based Testing of applications featuring dynamic instanciation of GUI widgets.

A. Preliminary activities

1) Definition of Test Objectives

According to the Software Engineering Body of Knowledge [21], *testing can be aimed at verifying different properties. Test cases can be designed to check that the functional*

specifications are correctly implemented, which is variously referred to in the literature as conformance testing, correctness testing, or functional testing. However, several other non-functional properties may be tested as well—including performance, reliability, and usability, among many others. Defining such test objectives is standard practice in software engineering. By building models prior to development, our approach is mainly geared towards Acceptance/Qualification Testing. Yet, one can elect our approach for the identification of faults in Reliability evaluation.

2) Modelling of the AUT and Definition of Rendering and Activation Functions

The production of a description of the AUT is a key element in our process as this description is used in both test generation and development activities. For the *Modelling of the AUT and Definition of Rendering and Activation Functions* activity (top-left of Fig. 6), a formal model engineer is appointed with the task of producing an ICO description of the application behavior in the CASE tool supporting the notation, Petshop. This model must be accompanied by the description of the rendering and activation functions, i.e. the mapping between GUI elements properties and information associated with places (token entered, token removed) and transitions (transition available or not) of the Petri net model.

B. Test generation activities

The test generation activities aims at producing, from the ICO description of the AUT, test cases and test scripts for the targeted implementation.

1) Definition of Test Selection Criteria

The Definition of Test Selection Criteria consists into translating the Test Objectives into constraints on the models for the test cases generation. For instance, in a scenario where the objective is *Acceptance/Qualification Testing*, the goal is to identify relevant places/transitions *for verifying that customer's requirements are met*. For instance, with the requirement "all control orders finish by either Abort or Send", we want the test case generator to produce test cases from the model by selecting paths on which the events related to the "Abort" and/or "Send" command are raised.

2) Simulation and Logging of Test Cases

The Simulation and Logging of Test Cases happens in the CASE tool supporting the ICO notation, Petshop. The aim of this step is to produce traces of an automated simulation (constrained by the test selection criteria) of the Petri net that contains:

- the invocations and events to be played on the SUT;
- the expected state of the GUI elements before/after the invocation/event is done/raised.

The key at this step is the production of relevant invocation/events, which can be made difficult by the parametrized nature of some of them. Indeed, while a click on a button is not parameterized, producing an input on a text box is, as it requires the actual string input. In a similar way, the simulator must know parameters of an invocation. Our strategies for dealing with these issues are the following:

- Event parameters are obtained by solving the pre-condition associated with the transition receiving the event using the Z3 SMT (Satisfiability Modulo Theory) solver [29] interfaced with Petshop;
- Instantiation and invocation parameters are provided manually by the user and may either be described as unique value, ranges (e.g. parameter *altitude* $\in$ *[0,25000]*) or arrays (e.g. parameter *flightNumber* $\in$ *("U2 9876", "NK 1234", "D7 6543")*).

At the end of the simulation, the logs are saved in an XML file. Fig. 7 provides an overview of the XSD associated with this XML file. It shows that a test description (testdesc) files contains:

- A definition of the User Interface (uidef) with the list of Petri nets involved in the simulation (obcses) as well as the events (event transition), invocations and renderings (places) they contain;
- A list of all the user interface states (uistates) encountered during the simulation, an uistate containing the parameters for activations (was a transition armed or not?) and renderings (what were the value carried by the tokens in a place?) functions.
- A list of test cases composed of test steps. Each test step refers to a uistate that must be verified before performing any actions. The actions in a step can either be invocations or production of events. After performing these actions, non-deterministic behavior of some GUI applications may lead to different uistates depending of the execution. To cope with that, Petshop logs all the possible nextsteps containing an UI state the GUI application may reach.

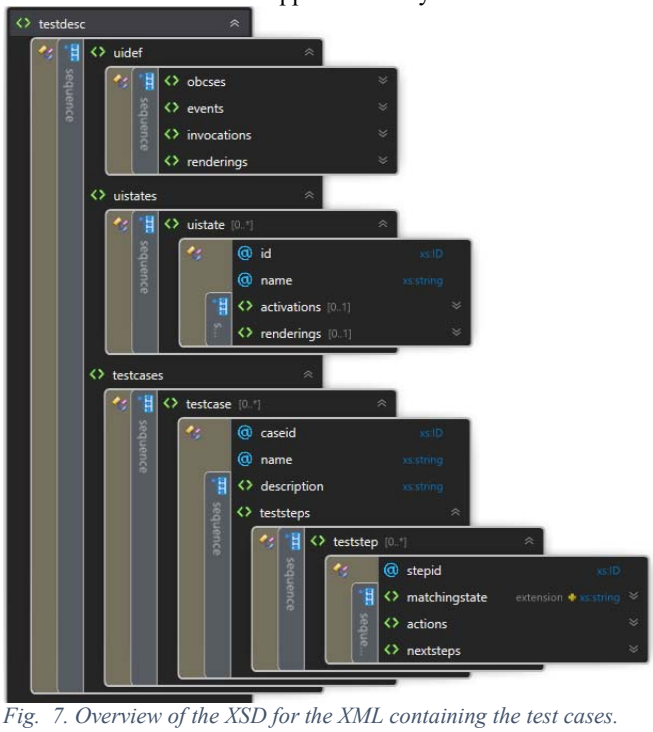

Fig. 7. Overview of the XSD for the XML containing the test cases.

3) Test Scripts Instanciation

The *Test Scripts Instantiation* consist is a semi-automated process that builds the structure of a test script (Junit, MStest, XCTest, etc.) from three elements:

- The test cases based on the results of the simulation;
- The UI Content description (name of widgets, containers, etc.);
- The list of rendering and activation functions from the ICO description.

For the test script generation to occur, the user must manually specify the actual correspondence between the renderings/activation functions and the application source code. Note that one may have decided to model the application with a particular target platform in mind (e.g. Java Swing), thus directly mentioning them in the ICO rendering and activation function. Yet, we claim that these functions should be generically described (e.g. "Deactivation of the send button") so test script can be instantiated for different target platform.

C. Development activities

1) Development of the AUT

The *Development of the AUT* consists in the production, by a team of software engineers and developers, of an implementation of the AUT for the targeted platform. Details on implementing from an ICO specification are provided in [32] and [4]). This process involves obtaining the reachability graph of the ICO Petri net. A reachability graph of a Petri-net is a directed graph G=(V,E), where $v \in V$ represents a class of reachable markings; $e \in E$ represents a directed arc from a class of markings to another class of markings [41]. We take advantage of the APT (Analysis of Petri nets and labelled transition systems) project to generate this graph [6].

D. Test Execution

The *Test Execution* is, due to the nature of the script we generate (Junit, MStest, XCTest, etc.), totally dependent of the IDE the testers are working on. For this reason we will not comment further on test execution.

VII. Conclusion and Future Work

Model-Based Testing requires modelling techniques that are expressive enough to describe the behavior of the Application Under Test [10][38]. Since long known for their complexity in the field of GUI Testing [2][30], GUI applications are getting more and more complex, featuring dynamic instantiation of widgets and complex interaction (e.g. multi-touch ones) [10][25]. This increasing complexity affects the ability of dynamically building model of the AUT by exploring its execution space, as done in the field of GUI Testing [37]. To cope with the increasing complexity, this paper proposed to build the application models from requirements as done in the standard Model-Based Testing process [38]. Comparing over 20 UIDLs, we have established that ICO [18] is one of the most suitable notation for describing the large set of behavior found in modern GUI applications. We demonstrated some of the ICO abilities by describing an Air Traffic Control application featuring dynamic instantiation of widgets as well as non-standard behavior (e.g. on pop-up menu and menu items).

The expressiveness of ICO enables the models to be used for two different purposes. On one hand, they are suitable as specification for the development of the application [4][32], on the other they enable test cases generation. In this paper, we proposed a development/testing process that takes advantage of both of these characteristics. While one may decide to model the application with a particular development environment in mind, we propose to rely on the generation of abstract test cases then instantiated for the targeted platform by referring to the ICO activation and rendering functions.

The approach proposed in this paper ends up with enabling the testing of the GUI applications using test scripts. Thus, we are conducting "offline" testing [38]. While this allow for easy replaying of test cases, this also implies that we constrain the test case generation so they are manageable in size and generation is not too long. We are currently investigating the opportunity of conducting "online" testing [38] using ICO. With online testing, the test generation algorithms can react to the actual outputs of the SUT. This has the potential to enable longer test run while benefiting from the correctness of models built from the application requirements.

References

[1] C. Appert and M. Beaudouin-Lafon, "SwingStates: adding state machines to the swing toolkit," in Proceedings of the 19th annual ACM symposium on User interface software and technology, Montreux, Switzerland, 2006, pp. 319–322, doi: 10.1145/1166253.1166302.

[2] I. Banerjee, B. Nguyen, V. Garousi, and A. M. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," Information and Software Technology, vol. 55, no. 10, pp. 1679–1694, Oct. 2013, doi: 10.1016/j.infsof.2013.03.004.

[3] E. Barboni, S. Conversy, D. Navarre, and P. Palanque, "Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification," in Interactive Systems. Design, Specification, and Verification, Berlin, Heidelberg, 2007, pp. 25–38, doi: 10.1007/978-3-540-69554-7_3.

[4] R. Bastide, D. Navarre, and P. Palanque, "A Model-based Tool for Interactive Prototyping of Highly Interactive Applications," in CHI '02 Extended Abstracts on Human Factors in Computing Systems, New York, NY, USA, 2002, pp. 516–517, doi: 10.1145/506443.506457.

[5] R. Bastide, D. Navarre, P. Palanque, A. Schyn, and P. Dragicevic, "A model-based approach for real-time embedded multimodal systems in military aircrafts," in Proceedings of the 6th international conference on Multimodal interfaces, State College, PA, USA, 2004, pp. 243–250, doi: 10.1145/1027933.1027974.

[6] E. Best and U. Schlachter, "Analysis of Petri Nets and Transition Systems," Electron. Proc. Theor. Comput. Sci., vol. 189, pp. 53–67, Aug. 2015, doi: 10.4204/EPTCS.189.6.

[7] R. A. Bolt, "'Put-that-there': Voice and gesture at the graphics interface," in Proceedings of the 7th annual conference on Computer graphics and interactive techniques, Seattle, Washington, USA, 1980, pp. 262–270, doi: 10.1145/800250.807503.

[8] J. C. Campos et al., "A More Intelligent Test Case Generation Approach Through Task Models Manipulation," Proc. ACM Hum.-Comput. Interact., vol. 1, no. EICS, pp. 9:1–9:20, Jun. 2017, doi: 10.1145/3095811.

[9] A. Canny, E. Bouzekri, C. Martinie, and P. Palanque, "Rationalizing the Need of Architecture-Driven Testing of Interactive Systems," in Human-Centered and Error-Resilient Systems Development, 2018.

[10] A. Canny, D. Navarre, J.C. Campos and P. Palanque, "Model-Based Testing of Post-WIMP Interactions Using Object Oriented Petri-nets." in 8th Formal Methods for Interactive Systems Workshop (FMIS).2019.

[11] L. Cardelli and R. Pike, "Squeak: a language for communicating with mice," SIGGRAPH Comput. Graph., vol. 19, no. 3, pp. 199–204, Jul. 1985, doi: 10.1145/325165.325238.

[12] L. Cheng, J. Chang, Z. Yang, and C. Wang, "GUICat: GUI testing as a service," in 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 858–863.

[13] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 429–440, doi: 10.1109/ASE.2015.89.

[14] P. Dragicevic and J.-D. Fekete, "Support for input adaptability in the ICON toolkit," in Proceedings of the 6th international conference on Multimodal interfaces, State College, PA, USA, 2004, pp. 212–219, doi: 10.1145/1027933.1027969.

[15] B. Dumas, D. Lalanne, and R. Ingold, "HephaisTK: a toolkit for rapid prototyping of multimodal interfaces," in Proceedings of the 2009 international conference on Multimodal interfaces, Cambridge, Massachusetts, USA, 2009, pp. 231–232, doi: 10.1145/1647314.1647360.

[16] J. Gregory, L. Berthoud, T. Tryfonas, A. Rossignol, and L. Faure, "The long and winding road: MBSE adoption for functional avionics of spacecraft," Journal of Systems and Software, vol. 160, p. 110453, Feb. 2020, doi: 10.1016/j.jss.2019.110453.

[17] A. Hamon, P. Palanque, J. L. Silva, Y. Deleris, and E. Barboni, "Formal Description of Multi-touch Interactions," in Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, New York, NY, USA, 2013, pp. 207–216, doi: 10.1145/2494603.2480311.

[18] A. Hamon, P. Palanque, J. L. Silva, Y. Deleris, and E. Barboni, "Formal Description of Multi-touch Interactions," in Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, New York, NY, USA, 2013, pp. 207–216, doi: 10.1145/2494603.2480311.

[19] A. Hamon, P. Palanque, M. Cronel, R. André, E. Barboni, and D. Navarre, "Formal Modelling of Dynamic Instantiation of Input Devices and Interaction Techniques: Application to Multi-touch Interactions," in Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, New York, NY, USA, 2014, pp. 173–178, doi: 10.1145/2607023.2610286.

[20] K. Hinckley, M. Czerwinski, and M. Sinclair, "Interaction and modeling techniques for desktop two-handed input," in Proceedings of the 11th annual ACM symposium on User interface software and technology, San Francisco, California, USA, 1998, pp. 49–58, doi: 10.1145/288392.288572.

[21] IEEE Computer Society, P. Bourque, and R. E. Fairley, Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.

[22] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems," Int J Softw Tools Technol Transfer, vol. 9, no. 3, pp. 213–254, Jun. 2007, doi: 10.1007/s10009-007-0038-x.

[23] D. Kammer, J. Wojdziak, M. Keck, R. Groh, and S. Taranko, "Towards a formalization of multi-touch gestures," in ACM International Conference on Interactive Tabletops and Surfaces, Saarbrücken, Germany, 2010, pp. 49–58, doi: 10.1145/1936652.1936662.

[24] K. Katsurada, Y. Nakamura, H. Yamada, and T. Nitta, "XISL: a language for describing multimodal interaction scenarios," in Proceedings of the 5th international conference on Multimodal interfaces, Vancouver, British Columbia, Canada, 2003, pp. 281–284, doi: 10.1145/958432.958483.

[25] V. Lelli, A. Blouin, B. Baudry, and F. Coulon, "On model-based testing advanced GUIs," in 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2015, pp. 1–10, doi: 10.1109/ICSTW.2015.7107403.

[26] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications," in Verification and Validation 2012 IEEE Fifth International Conference on Software Testing, 2012, pp. 81–90, doi: 10.1109/ICST.2012.88.

[27] A. M. Memon and B. N. Nguyen, "Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End," in Advances in Computers, vol. 80, M. V. Zelkowitz, Ed. Elsevier, 2010, pp. 121–162.

[28] 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., vol. 16, no. 4, pp. 18:1–18:56, Nov. 2009, doi: 10.1145/1614390.1614393.

[29] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, 2008, pp. 337–340, doi: 10.1007/978-3-540-78800-3_24.

[30] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," Autom Softw Eng, vol. 21, no. 1, pp. 65–105, Mar. 2014, doi: 10.1007/s10515-013-0128-9.

[31] S. Oney, B. Myers, and J. Brandt, "ConstraintJS: programming interactive behaviors for the web by integrating constraints and states," in Proceedings of the 25th annual ACM symposium on User interface software and technology, Cambridge, Massachusetts, USA, 2012, pp. 229–238, doi: 10.1145/2380116.2380146.

[32] P. A. Palanque, R. Bastide, L. Dourte, and C. Sibertin-Blanc, "Design of user-driven interfaces using Petri nets and objects," in Advanced Information Systems Engineering, Berlin, Heidelberg, 1993, pp. 569–585, doi: 10.1007/3-540-56777-1_30.

[33] P. Palanque, R. Bastide, and F. Paternò, "Formal Specification as a Tool for Objective Assessment of Safety-Critical Interactive Systems," in Human-Computer Interaction INTERACT '97: IFIP TC13 International Conference on Human-Computer Interaction, 14th–18th July 1997, Sydney, Australia, S. Howard, J. Hammond, and G. Lindgaard, Eds. Boston, MA: Springer US, 1997, pp. 323–330.

[34] P. Palanque, R. Bernhaupt, D. Navarre, M. Ould, and M. Winckler, "Supporting Usability Evaluation of Multimodal Man-Machine Interfaces for Space Ground Segment Applications Using Petri nets Based Formal Specification," in SpaceOps 2006 Conference, 0 vols., American Institute of Aeronautics and Astronautics, 2006.

[35] P. Palanque, J.-F. Ladry, D. Navarre, and E. Barboni, "High-Fidelity Prototyping of Interactive Systems Can Be Formal Too," in Human-Computer Interaction. New Trends, Berlin, Heidelberg, 2009, pp. 667–676, doi: 10.1007/978-3-642-02574-7_75.

[36] C. A. Petri, "Communication with automata," http://edoc.sub.uni-hamburg.de/informatik/volltexte/2010/155/pdf/diss_petri_engl.pdf, 1966.

[37] M. Pezzè, P. Rondena, and D. Zuddas, "Automatic GUI Testing of Desktop Applications: An Empirical Assessment of the State of the Art," in Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, New York, NY, USA, 2018, pp. 54–62, doi: 10.1145/3236454.3236489.

[38] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Softw. Test. Verif. Reliab., vol. 22, no. 5, pp. 297–312, Aug. 2012, doi: 10.1002/stvr.456.

[39] K. Vorobyov and P. Krishnan, "Comparing model checking and static program analysis: A case study in error detection approaches," Proceedings of SSV, 2010.

[40] T. E. J. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "TESTAR: Tool Support for Test Automation at the User Interface Level," IJISMD, vol. 6, no. 3, pp. 46–83, Jul. 2015, doi: 10.4018/IJISMD.2015070103.

[41] X. Ye, J. Zhou, and X. Song, "On reachability graphs of Petri nets," Computers & Electrical Engineering, vol. 29, no. 2, pp. 263–272, Mar. 2003, doi: 10.1016/S0045-7906(01)00034-9.