Model-Based Testing of Interactive Systems

Ankur Saini

Internship Report

Model-Based Testing of Interactive Systems

Author: Ankur Saini[1]
Supervisors: Arnaud Blouin[2], Benoit Baudry[3]
Laboratory: INRIA, Rennes
Team: INRIA/Triskell

Master Research - Software and Formal Methods (SOCM)
TELECOM Bretagne, Brest

June 02, 2011

Abstract: *In the current era every interactive system provides User Interfaces (UIs) or interaction features. WIMP (Window, Icon, Menu, Pointing device) denotes interactive systems based on mouse and keyboard interactions to manipulate widgets such as windows, menus and dialogue boxes. Post-WIMP interactive systems aim at going beyond these concepts to notably reduce the gap between the user and the system. An extensive research on model-based testing of interactive systems has already been done, but all are limited to WIMP interactive systems. This report introduces a novel model-based testing approach for post-WIMP interactive systems. We extend the models provided by the Malai architecture to build test models. Three strategies have been implemented to generate test cases automatically from the test model. Finally, we apply the novel approach to the open-source post-WIMP interactive system LaTeXDraw[4] to validate the ability of generated test cases to reveal bugs.*

[1]Ankur.Saini@telecom-bretagne.eu, TELECOM Bretagne, Brest
[2]Arnaud.Blouin@inria.fr, INRIA, Rennes
[3]Benoit.Baudry@inria.fr, INRIA, Rennes
[4]`http://latexdraw.sourceforge.net/`

Contents

List of Figures

1 INTRODUCTION

Contents

1.1 Interactive system

*"**Interactive systems** are computer systems that require interaction from the user to perform actions"* [10]. The direct communication between the user and the system occurs *via* User Interfaces (UIs) and interactions.

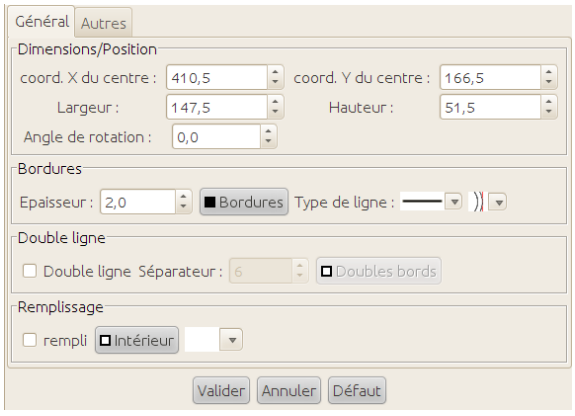


Figure 1.1: WIMP Interactive system

WIMP (Window, Icon, Menu, Pointing device) denotes interactive systems *"based on a single mouse and keyboard to manipulate windows, icons, menus, dialogue boxes, and drag and drop objects on the screen"* [8]. WIMP system's UIs are composed of graphical components, called widgets, such as buttons and check-boxes. For example, Figure 1.1 illustrates a formular using widgets.

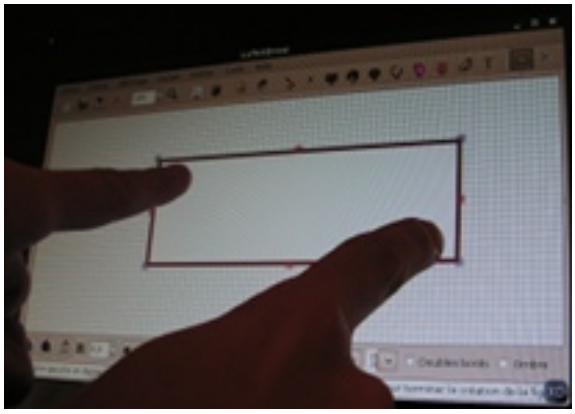


Figure 1.2: Post-WIMP Interactive system

Post-WIMP interactive systems aim at going beyond the concepts of WIMP. *"A post-WIMP interface is one containing at least one interaction technique not dependent on classical 2D widgets such as menus and icons"* [42]. Post-WIMP interactive systems focus on interactions carried out by users. For instance, smart phones provide users with bi-manual, gyroscopic, and multi-modal (*e.g.* combining voice with gesture) interactions. Another kind of post-WIMP interactive systems are virtual reality systems where the UI is a virtual representation of a location. For example, Figure 1.2 illustrates the bimanual interaction used to scale shapes.

1.2 Motivation

All systems provide users with UIs or interaction features. UIs make testing more difficult of many reasons: unwanted events (*e.g.* a user gives printing command from his application, but during printing operating system pops up a dialog box showing printer is out of paper), many ways in/out (*e.g.* a same action could be carried out using a keyboard shortcut, a menu option, a button click, *etc.*), infinite input domain (*e.g.* a user having complete freedom has 120 ways to enter data into five text fields (factorial of five)), and hidden synchronization (*e.g.* a text field to enter espouse name is visible if a user selects the radio button *Couple*, otherwise the text field is invisible). Because of these problems, testers cannot test every combination of events that a UI must support.

In manual testing of interactive systems, testers examine requirements of the system (*e.g.* a given text field only accepts numeric values), create test cases (*e.g.* choose correct/incorrect values for the text field) and execute them (*e.g.* enter the chosen values in the field) [37]. Observed and expected behaviours are compared in order to yield the test verdicts whether the system is working as intended. Manual testing is time and effort consuming. In several cases, manual testing is the recommended way to test certain functionalities. This approach is not exhaustive since human thoroughness may not be sufficient to test the system.

An attractive approach to automate test execution is *capture and replay* [9]. It is treated as an easy way to produce test scripts. Capture and replay tools (CRTs) such as *IBM Rational Robot*, *Mercury WinRunner*, *CompuWare TestPartner* and *Segue's Silk-*

Test [20] can be used to automatically record test scripts. While the tester interacts with the system under test (SUT) these tools capture the interactions and record the UI events/actions in test scripts. The recorded information is usually positional (*e.g.* click on the button A at the screen coordinates X, Y). These recorded test scripts can then be (re-)executed on the SUT when needed in order to yield test verdicts by comparing with expected captured screen. The limitation of this approach is the cost of script maintenance and script recording: whenever there are changes in UIs components, test steps affected by the changes may need to be re-captured and updated in existing tests.

During regression testing of interactive systems, more than 74% of the test cases become unusable [32]. Internal assessment in *Accenture* shows 30% to 70% test cases need to be changed in the test scripts [19]. A number of survey results exhibit model-based testing as a promising solution to overcome the limitation of capture and replay approach [15]. *Model-based testing* facilitates the testing of a system against an abstract model of what it should be. It permits automating test case generation and test case execution. Automated test case generation offers to reduce testing cost [16]. A model, built manually or automatically, is used to generate test cases. Since it covers a lot of test cases automatically, model-based testing is effective for testing of interactive systems [22]. A UI is represented as a set of objects, a set of properties and a set of events that change the properties of certain objects. So a model can be produced by a UI representation based on some coverage criteria. Once test cases are generated, test case execution can be automatically done and the current output is compared with the expected output (the oracle) to make test verdicts. The limitation of model-based testing approaches is that they are limited to the study of WIMP interactive systems: they only focus on UIs and their components but not on the interactions performed by users to interact with UIs, nor the effects of these interactions on the underlying system.

1.3 Internship objective

Post-WIMP interactive systems do not consider widgets as first-class components but prefer the use of concepts that reduce the gap between the user and the system. For instance, direct manipulation is a principle stating that the users should interact through the UI with the common representation of the manipulated data [41]. A classical example of direct manipulation UI is a vectorial shape editor in which UI presents the 2D/3D representation of the shapes to handle. So the problem of testing post-WIMP systems is that they do not depend mainly on widgets nor a graph of events applied on widgets. For example a recognition-based interface, where the input is uncertain, *"a conventional event-based model may no longer work, since recognition systems need to provide input continuously, rather than just in discrete events when they are finished"* [34]. So the testing of post-WIMP systems has to mainly focus on interactions rather than UIs' component. In [7], Beaudoin-Lafon explained that developers have to switch from designing UIs to interactions. This principle should also be applied on UI testing. Moreover, current UI testing approaches do not check what the interactions do on the data of the system. A modern UI testing approach must also provide features to test that.

The main objective of this internship is to explore related work on testing approaches and propose a model-based approach to test post-WIMP interactive systems. For the testing, we will do the classification of existing post-WIMP interactive systems and the

determination of test objectives of the intended class of the systems. One architecture of post-WIMP interactive systems will be selected among several existing architectures. Finally, the novel testing approach will be applied on an open-source project to validate the ability of generated test cases to reveal bugs in post-WIMP interactive systems.

2 STATE OF THE ART

Contents

Inspite of extensive research on model-based testing of interactive systems, most studies address to WIMP UIs where the focus is on the components (widgets) of the UIs. Whereas, post-WIMP UIs consider interactions as first class objects.

Model-based testing is an approach that uses the concept of *model*. The general process of model-based testing is illustrated in Figure 2.1. The model is usually an abstract presentation of the behaviour of the system under test. The model is built from specifications or requirements of an interactive system either manually or automatically. The model is then used to generate test cases. Generated test cases are executed on the system which gives actual behaviour. The evaluation of test results is done by comparing the intended (*i.e.* test oracle) and actual behaviours. This chapter explains the fundamental terms of model-based testing of interactive systems in Section 2.1 along with its state of the art and three existing architectures of post-WIMP interactive systems in Section 2.2.

2.1 Model-based testing

A number of survey results exhibit model-based testing as a promising solution to overcome the limitation of capture and replay approach [15]. It helps to reduce the time and effort on testing without compromising the quality and the maintenance of the tests. Model-based testing facilitates the testing of a system against an abstract model of what

it should be. It permits automating test case generation and test case execution. Since it covers a lot of test cases automatically, model-based testing is effective for the testing of interactive systems [22].

The fundamental terms of model-based testing are explained in Section 2.1.1. Section 2.1.2 addresses some existing approaches of model-based UI testing.

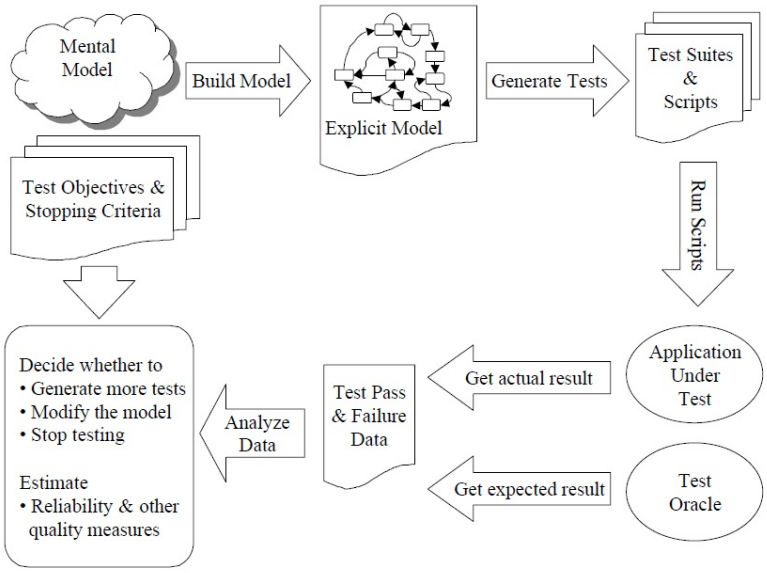


Figure 2.1: Model-Based Testing Process [17]

2.1.1 Fundamental terms

This section explains the fundamental terms of the model-based testing of interactive systems.

Model

An abstract *model* is the basic requirement in model-based testing built from specifications or requirements of an interactive system. An abstract model represents intended behaviour of a system under test (SUT). A model can be built before or parallel to the development process of a SUT. It can also be constructed from a completed system (*e.g.* from source code). A model is created manually (doing by experts) or automatically (doing by model transformation). The big challenge in model-based testing is the creation of abstract models. Different notations have been suggested for expressing models having varying degrees of formality. They range from textual form written in *CNL* [13] to entirely formal specifications written in *Z*, as well as abstract *state machines* [29] and *graph structures* [38]. Models can be carried out in different level of abstractions. For example, Nguyen et al. [37] use two models: action model and machine model, possessing different levels of abstraction. Action model defines abstract actions (*e.g.* save as a file). Machine model defines sequence of concrete events of each abstract action (*e.g.* click on menu file, click on menu item save as and so on). If any changes occur in UIs, only machine model's events are needed to be changed, while changes in action model are very rare.

ConcurTaskTrees (CTT) is a notation for task models which are represented as tree structure [38]. A task is an activity performed to reach a particular goal. Task models capture complete set of user interaction tasks and define how the tasks are inter-related.

SpecExplorer [39] tool employs a modelling language "Spec#". A graphical front-end is provided which allows testers to describe UI behaviour in UML diagrams. This behaviour is then transformed into a Spec# program containing action functions (events). Later testers fill each action's body to define semantics of them.

Some other existing models are *use case model* that expresses functional requirements [9]; *hierarchical predicate transition net* that represents UI's structure [40]; *event-flow model (EFG)* that represents UI's components [29].

Test suite generation

Test coverage criteria play a vital role in model-based testing in two ways. First, they are defined to constitute an adequate test suite (*i.e.* what to test in an interactive system). Second, they help to determine whether the generated test suite adequately tests the implementation or not. Test coverage criteria are set of rules to partially describe the behaviour of a model. In certain testing problems, user intervention is needed specifying what to test in a UI [30]. Some common examples of coverage criteria in structural based testing of a program are statement coverage, branch coverage and path coverage. Using source code, they respectively require execution of each statement, branch and path. Because source code is not always available, these criteria should not always be based on source code. For example, Memon *et al.* [33] define a new class of coverage criteria "event-based coverage criteria" (*e.g.* event coverage, event-interaction coverage, length-n event sequence coverage and so on) in terms of UI events and their interactions within a component or among components. For example, event coverage requires each event in the component to be performed at least once. Whereas, event-interaction coverage checks the interactions among all possible events in the component.

Test cases are derived from an abstract model by taking coverage criteria into account. A test case is a sequence of individual steps (*e.g.* sequence of mouse/keyboard events). Two types of testing are possible: on-line testing and off-line testing. In off-line testing, test cases are generated strictly before they are executed on a SUT. Whereas in on-line testing, some test cases are generated on-the-fly. There are several approaches to automatically generate test cases. For example, a sequence of UI events can be generated randomly. On the other hand, based on initial and final goals, a shortest path search algorithm can be used on a state-transition based model to generate a sequence from initial goal to final goal. Since, coverage criteria are used to generate test cases, thus adequacy of the generated test suite depends on maturity of them. Generated test cases are always at the same level of abstraction as the given model. Memon *et al.* [30] presented a test case generation system called "*planning assisted tester for graphical user interface systems (PATHS)*" that uses AI planning. The Tester provides a specification of initial and goal states for commonly used tasks. PATHS generates plans for each specified task. Each generated plan represents a test case because it reflects an intended use of the system.

The main drawback that model-based testing tools have to deal with is state **explosion problem**: the number of states in a SUT can be so large that the SUT becomes impractical to analyse. So, test case generation has to be controlled to generate test cases

of manageable size while still guaranteeing adequate testing. A generated test case may have its length variable. Yuan *et al.* [45] show that test cases with certain length may have higher defect detection ability. Memon *et al.* [43, 21] show that long test cases are better than short ones in defect detection, but the number of sequences grows exponentially with length. A genetic algorithm is also used to repair infeasible UI test suites. A test case is infeasible if at least one of its events that is expected to be available during execution is unavailable.

Test oracle

Test oracle is a mechanism for comparing the intended behaviour with the actual behaviour of a SUT to provide *test verdicts* (*i.e.* pass or fail). It determines whether a SUT behaves correctly when a test case is executed on it. A test case execution passes if the intended behaviour and its actual behaviour are equals. Memon *et al.* [31] presented an approach of automated test oracles for UIs. This approach has the following three main components: expected-state generator, execution monitor and verifier. An expected-state generator uses the UI representation to automatically derive the UI's expected state for each test case. Execution monitor provides UI's actual state to the oracle. Finally, a verifier in the oracle automatically compares the two states to provide test verdict. It compares both UI states after each execution of an event in a sequence of events.

Test suite execution

The execution of generated test cases could be manual or automatic. Testers can execute test cases manually directly on the SUT. Whereas, automatic execution cannot be done directly on a system under test (SUT) because of their abstraction. Their execution requires the creation of concrete test cases from abstract test cases. An *adaptor* is a program used to simulate user actions on a SUT. Whenever, a SUT is to be tested, generated test suite is given as input to its adaptor which simulates them.

2.1.2 Model-based testing approaches

Some existing model-based testing approaches for interactive systems are described as follows.

Memon *et al.* [27, 28, 29] provide a *GUI testing framework "GUITAR"* that includes an event-based modeling method. Since a UI contains large number of widgets expecting events, the UI is decomposed into components. Each component is defined in terms of modal windows. There are two kinds of windows: modal windows that restrict focus of interaction (*e.g.* "save as" window); modeless windows that do not restrict focus (*e.g.* "find" window). Each component is represented by an *Event flow graph (EFG)* containing of a set of events. Each node of EFG represents an event and a directed edge represents the follow relationship. *Integration tree (IT)* is modelled in case of inter-component interactions. Memon *et al.* [26] propose a reverse engineering tool "GUI Ripper" to automatically obtain EFGs and IT directly from the executable UI. *GUI ripping* is a dynamic process in which the system's UI is automatically traversed by opening all its windows and extracting all thier widgets, properties and values. Memon *et al.* also presented a test case generation system, "planning assisted tester for graphical user interface systems (PATHS)". Test case generation process consists of two phases: setup phase where testers have to define preconditions (*e.g.* content should be selected to use "cut" event) and effects (*e.g.* selected

content must be removed once the "cut" event is triggered) of all events; plan generation phase where tasks are identified by defining a set of initial and goal states, and test cases are auto-generated by chaining preconditions and effects between initial and goal states. Oracle is also implemented in this system as a component [31]. Betweenness clustering method is applied on EFG to partition UI regions. Network centrality measures are also applied to identify most important events and event sequences in UIs [18]. A genetic algorithm is used to repair infeasible test suites [21]. Test cases can be dynamically generated using feedback from run-time states [44].

Nguyen et al. [37] propose a framework "*Action-Event Framework (AEF)*". It is a two layered approach: top layer is action model which represents business logic and it defines abstract actions (e.g. open a file); bottom layer is mapping model which represents presentation logic and it defines mapping of each abstract action to its sequence of concrete UI events (e.g. click on menu "file", click on menu item "open", and so on). The action model can be used with multiple mapping models. There is no need to create a new action model for each variation of the UI. Using of abstract actions, any change in the presentation logic will only affect the mapping model not the action model. A GUI test generator (GTG) [36], guided by coverage criteria (structure-based coverage criteria - state coverage and transition coverage) is used to generate test cases.

Bertolini et al. [9] propose a framework which uses UI *use cases* as a model. Use cases are written in controlled natural language (CNL) [13]. CNL is a precise subset of English where verbs are actions and nouns are component names. The TaRGeT (Test and Requirements Generation Tool) tool selects test case generation algorithms and use cases to generate test suite.

Kervinen et al. [22] propose a manual modeling of *action machine* and *refinement machine* that represent business logic and implementation logic respectively. Both machines are represented as labelled transition systems (LTSs). High level events are action words (e.g. open a file) represented in action machine and low level events are keywords (e.g. click the OK button) that implement the actions words represented in refinement machine. To generate test cases, each action in the action machines is replaced by its corresponding refinement machines in order to obtain composite LTS, and test cases are generated from it. The advantages of this approach is that it allows to reuse its LTSs. For example, during UI changes, all we need to change in its refinement machine. In the case where a feature in a system is not being tested, it is enough to remove the corresponding action words from the action machine.

2.2 Interactive system architectures

This section describes three existing architectures of post-WIMP interactive systems.

2.2.1 Malai

Malai is an architectural model for interactive systems [11]. In Malai, a UI is composed of presentations and instruments (see Figure 2.2). A presentation is composed of an abstract presentation and a concrete presentation. An abstract presentation is a representation of source data created by a Malan mapping (link ①). A concrete presentation is the graphical representation of the abstract presentation. It is created and updated by another Malan

mapping (link ②) [12]. An interaction consumes events produced by input devices (link ③). Instruments transform input interactions into output actions (link ④). The interim feedback of an instrument provides users with temporary information that describe the current state of interactions and actions they are carrying out (link ⑤). An action is executed on the abstract presentation (link ⑥); source data and the concrete presentation are then updated throughout a Malan mapping (link ⑦).

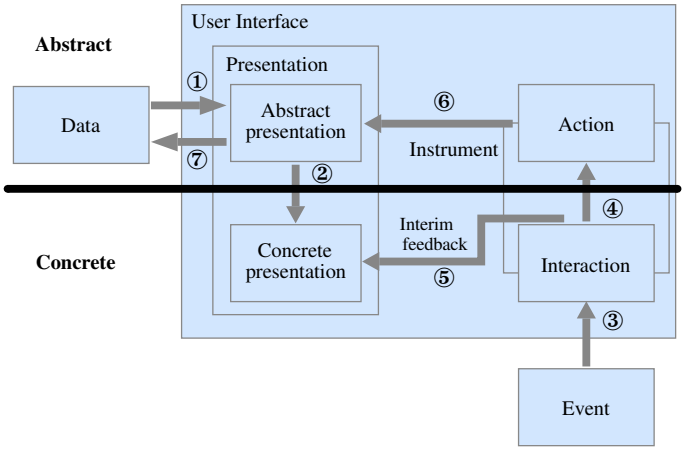


Figure 2.2: Organisation of the architectural model Malai [11]

Malai aims at improving: 1) the modularity by considering presentations, instruments, interactions, and actions, as reusable first-class objects; 2) the usability by being able to specify feedback provided to users within instruments, to abort interactions, and to undo/redo actions.

2.2.2 VIGO

VIGO stands for *Views, Instruments, Governors and Objects* [24]. VIGO architecture is designed for the implementation of multi-surface interactions. It is an extension of the instrumental interaction model [6] and MVC (Model-View-Controller) [25]. VIGO is designed to create distributed interfaces based on the principles of decoupling and integration. Decoupling means to separate objects and instruments that communicate through a simple protocol to manipulate the objects. Whereas, integration means the system should appears as a single entity where an interaction involves multiple surfaces, multiple processes and multiple machines from the user perspective.

VIGO architecture is illustrated in Figure 2.3. Objects are presented through views to the user and are manipulated through instruments. When an instrument manipulates an object, it queries the governors associated to that object to validate the manipulations. In order to achieve the first principle of decoupling between objects and instruments, objects are passive *i.e.* no operation or method is provided to them. They only provide direct accessible properties. The behaviour provided through operations or methods are encapsulated in governors. VIGO view is very close to MVC view. Views do not provide any kind of interaction: any change to the view reflects the change in the object. Beaudouin-Lafon defines an instrument as *"a mediator or two-way transducer between the user and domain objects. The user acts on the instrument, which transforms the user's actions into commands affecting relevant target domain objects"* [6]. The concept of instrument

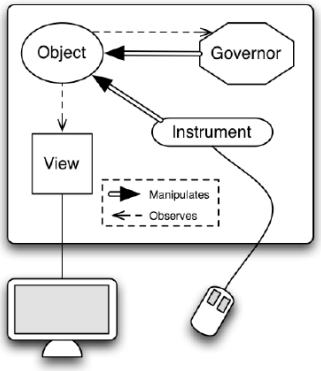


Figure 2.3: VIGO architecture [24]

is inspired by the way we use physical tools. For example, a pencil is used to write on a piece of a paper, a hammer to drive a nail. An interaction occurs through chains of instruments *e.g.* An instrument to select an object is chained with an instrument to move an object. Governors embody the rules and the consequences of interactions with objects and they implement application logic and business rules found in MVC. User manipulates an object through instruments and indirectly through governors. Governors are loosely coupled with objects so, they can be dynamically coupled and decoupled with objects.

2.2.3 ICOs

Interactive Cooperative objects (ICOs) is a model-based user interface description language (MB-UIDL) that provides a formalism of the dynamic behaviour of an interactive system. An ICO specification fully describes physical interactions (*e.g.* mouse behaviour). *"In ICO UIDL, an object is an entity featuring four components: a cooperative object which describes the behaviour of the object, a presentation part, and two functions (the activation function and the rendering function) which make the link between the cooperative object and the presentation part"* [35]. Figure 2.4 illustrates metamodel of the ICO UIDL.

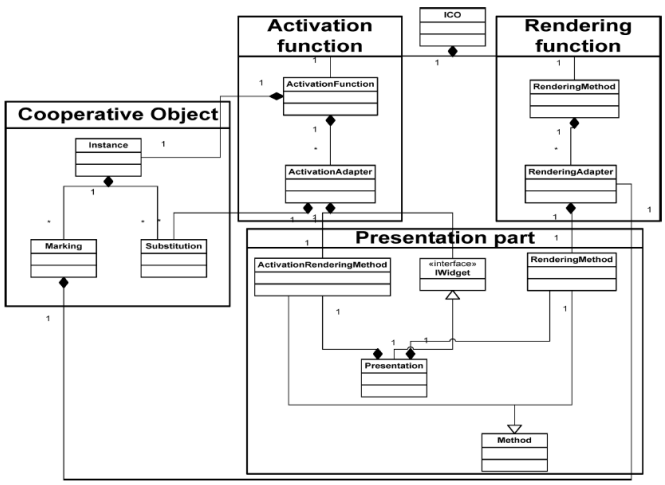


Figure 2.4: Metamodel of the cooperative object formalism [35]

The presentation part of an object corresponds to its external appearance (*e.g.* in WIMP UIs, a window containing a set of widgets). Cooperative object (CO) characterizes how the object reacts to the external stimuli according to the internal state (stimuli are either methods calls or event notifications). CO description is created by a software interface and its behaviour is expressed using Petri nets. Activation function states the relationship between user services and widgets, and it associates each event from the presentation part to the event handler to be triggered and to the corresponding rendering method for representing the activation and deactivation. Whenever a user trigger an event, the activation function is notified and it requires the CO to fire the corresponding event handler providing the value from the user event. The consistency between the internal state and its external appearance of the system is maintained by the rendering function by reflecting system states changes on the user interface.

2.3 Discussion

In this chapter, we have discussed the fundamental terms of model-based testing of interactive systems and three architectures of interactive systems. The model-based testing has many advantages: the support of both automated test case generation and test case execution; reduces the time and manual testing efforts; low maintenance cost. But the main limitation is that, it addresses to WIMP interactive systems. The evolution of systems, input devices and the ways to use these systems bring changes in the way to create UIs. Post-WIMP interactions and UIs are being commonly used now (*e.g.* bi-manual interaction on smart phones). So, the testing of these systems yields a new challenge to tackle.

3 CONTRIBUTION

Contents

To test post-WIMP interactive systems, an abstract model *i.e.* a test model that captures their behaviour is needed. For the model-based testing of such interactive systems, we need to understand what the systems do (Section 3.1). Once this is done, the test model from which test cases are generated can be built (Section 3.2). We are building the test model manually by extending the Malai models using *succeed* relation. To generate test cases automatically from the model, we describe algorithms using three different coverage criteria (all-path, all-transition, and all-state coverage) in Section 3.3. The execution of test cases is explained in Section 3.4.

3.1 System understanding

The most common requirement to test a system is to have good understanding of what the system does. It is essential to build a mental model that represents the system functionalities and use this information to build its abstract model. But this step is not trivial

since most of the current systems are being made having complex functionalities and interactions.

In order to understand systems, we need to learn about the systems and their environment as well. Doing the review of systems' requirements and specifications, we can accumulate all information necessary to build their abstract model. The procedure that we followed to understand post-WIMP interactive systems is explained as follows.

1. **Determine the class of system**.
   As computing devices (*e.g.* smart phones) evolve, the conception of post-WIMP interactive systems evolves as well. Nowadays many classes of post-WIMP interactive systems exist having varying degrees of functionalities and interactions. Augmented reality systems, virtual reality systems, tangible interaction, ubiquitous computing systems, and so forth are several examples of post-WIMP interactive systems. In our work, we decided to focus our work on conventional post-WIMP interactive systems such as *CPN2000*[1] [8] and *LaTeXDraw*[2]. We identify a conventional post-WIMP interactive system as a system that can be easily tackled, and having less functionalities and interactions than the systems aforesaid.

2. **Determine the test objectives**.
   For the testing of systems of the intended class, we need to determine which components or features are interesting to test and define them as test objectives. In [5], Bass *et al.* proposed an architectural model of interactive systems, called *Arch*. The *Arch* model is shown in Figure 3.1. The three right-hand side components of the architecture *i.e. physical interaction, logical interaction, and dialog controller* are the interaction part of the interactive systems. Whereas, the remaining two components are non-interaction part of the interactive systems. The *physical interaction* handles interactions provided by tools that a user carries out to execute actions, and renders information provided by the *logical interaction* component. The *logical interaction* transforms the information provided by the *physical interaction* component into abstract information, so that the transformed information is forwarded to the *dialog controller* component (and *vice versa*). The *dialog controller* assures the sequence of events and defines how they change the inner state of the system, and how the system reacts. The *functional core adaptor* translates the calls from *dialog controller* into functional core instructions (and *vice versa*). The *functional core* controls and executes actions on the objects.

   By considering these different components, we defined several test objectives, as explained below.

   - For the manipulation of an object, a tool (or an instrument) is needed. A tool is a mediator between a user and an object. For example, a pencil is needed to write text and a screwdriver is needed for doing/undoing a screw. So we need to test the selection of the tool *i.e.* whether the *dialog controller* component is working correctly.
   - In post-WIMP interactive systems, interactions are first-class objects. So the testing of interactions is a major activity *i.e.* whether the *logical interaction* component is working correctly.

[1] CPN2000: `http://cpntools.org/`
[2] LaTeXDraw: `http://latexdraw.sourceforge.net/`

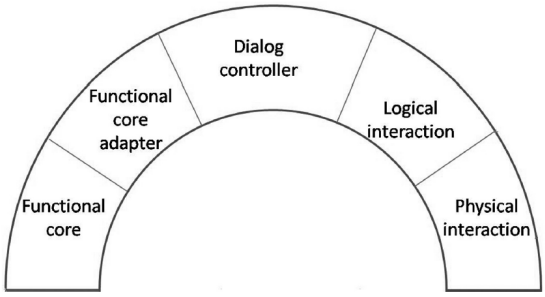


Figure 3.1: The Arch model [5]

- Since the input is uncertain, systems must include feedback to allow users for monitoring and correcting the interpretation. Consider an example of the creation of a circle in a drawing editor. A user clicks and drags the mouse, and a feedback is provided in the form of a rubber-band circle. When the button is released, the object is created. So the feedback plays a vital role in post-WIMP interactive systems and it is necessary to test the feedback *i.e.* whether the rendering performed by *physical interaction* component is working correctly.
- After manipulation of objects, the functional core data may be affected. So the testing is needed in order to check whether the data is affected.

3. **Explore some input sequences that would reveal bugs**.
   Prior to the building of a test model of a system, an exploration of its input sequences should be done manually. If the input sequences are able to reveal as much bugs as possible, then the test model should be built in such a manner that will generate all these sequences *i.e.* test cases. The test model may also generate other information such as oracle i.e. expected behaviour (if possible and needed) with the test cases. To test the intended systems, we explored several input sequences. Consider an example of *drag-and-drop* interaction to draw a circle in the *LaTeXDraw* interactive system. To draw a circle we follow a sequence of events such as: 1) press the mouse button to start the interaction; 2) drag the mouse (while dragging temporary feedback is provided by displaying a rubber-band circle); 3) Press the *ESCAPE* key to abort the interaction. After executing this sequence on the system, we obtained one bug that the abortion of the interaction by pressing the *ESCAPE* key does not work. Through this exploration, we identified that the expected test model should be able to generate these sequences.

This initial exploratory phase provides a clear understanding of post-WIMP systems and the specific testing tasks they requires. We now have a clear idea of the type of defects that must be targeted by test cases, as well as an intuition of how test cases can target these defects. On the basis of this knowledge, we propose a metamodel that specifies the structure of test models for post-WIMP systems. This metamodel aims at letting testers to capture all information needed for test generation in a global test model. After the understanding of the system under test, the next step is to build a model that let us to generate test cases as explained in the following section.

3.2 Model generation

System's specifications and supported interactions must be used to build a test model that captures the behaviour of a post-WIMP interactive system under testing. The representation of the test model must satisfy few requirements. First, the abstraction of the test model should be at high level to make it free from platform specific details. Second, it should be scalable and expressive enough to represent a wide range of interactions. Finally, it should contain low level details of interactions to develop a test oracle.

We have explained several interactive system architectures in Section 2.2. In Section 3.2.1, we identify several advantages that lead us to select Malai architecture for our testing purpose. The different models provided by Malai are illustrated in Section 3.2.2. We are building the test model manually by extending the Malai models using *succeed* relation, as explained in Section 3.2.3. Section 3.2.4 shows an example of the representation of a test model.

3.2.1 Advantages of Malai

We studied several architectures of interactive systems that may be used for the testing purpose. Because it has several advantages, we chose Malai architecture as explained in Section 2.2.1.

The ICOs architecture provides a formalism of the dynamic behaviour of an interactive system and defines the behaviour using Petri nets. But the definition of a component's behaviour is more complex in ICOs than Malai. Malai defines interactions based on low level events (*e.g.*"key pressed" or "key released"), whereas ICOs defines interactions based on high level events (*e.g.* "dragging" or "click events") consisting of low level events. Low level details provided by Malai may be used to partially define the test oracle (interactions' states). Moreover, ICOs does not provide any formalism neither for aborting actions/interactions nor interim feedback that plays a vital role in post-WIMP systems.

As explained in Section 2.2.2, VIGO is an instrumental-based architecture for the implementation of ubiquitous instrumental interactions to create distributed UIs. The testing of post-WIMP interactive systems requires the focus on all possible interactions *i.e.* user behaviours. VIGO does not separate actions, interactions and instruments concepts whereas, Malai separates them. These separated concepts may be extended to build the test model.

The models provided by Malai are explained in detail in the following section.

3.2.2 Model-based testing with Malai

For the testing of post-WIMP interactive systems, we are extending the different models provided by Malai. These models are illustrated as follows.

Interaction
An interaction model describes the behaviour of interaction using a finite state machine. A given condition should be respected in order to fire a transition. Here, a condition is composed of two parts: event name (*e.g. pressKey*); events' parameters (*e.g. key=='a'*). For example, the *pressKey*|*key=='a'* states that the state will be changed when a *pressKey*

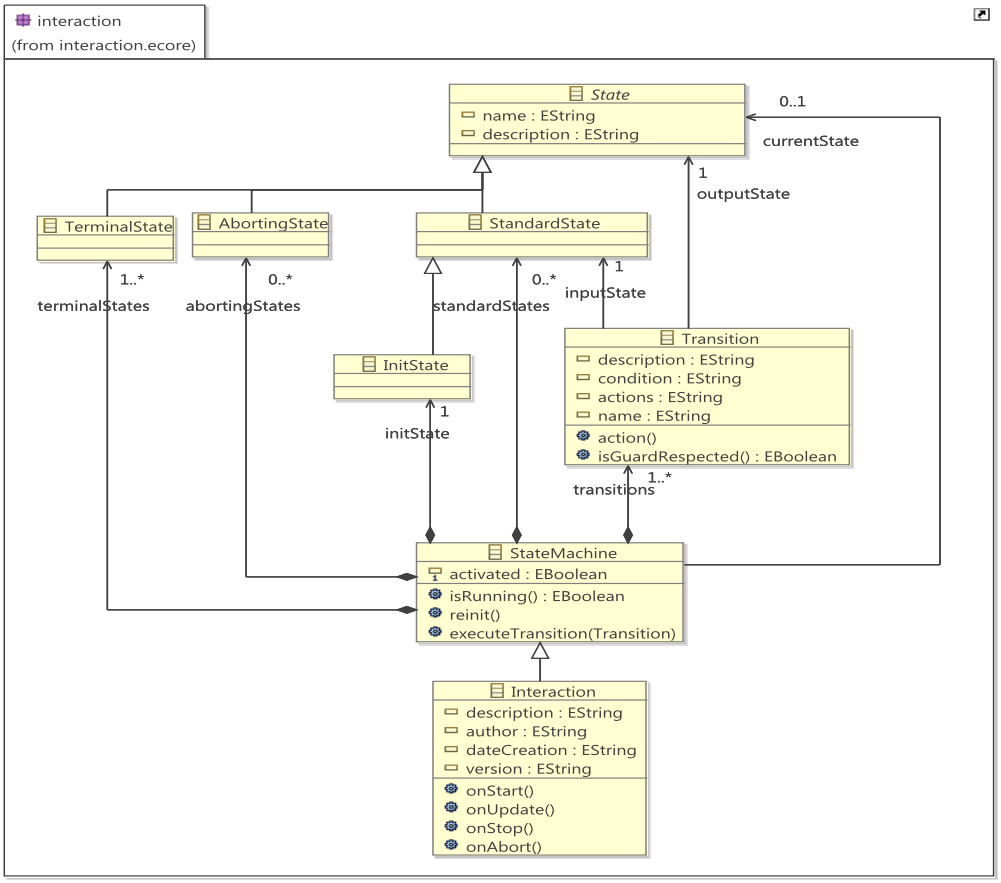


Figure 3.2: Interaction metamodel [10]

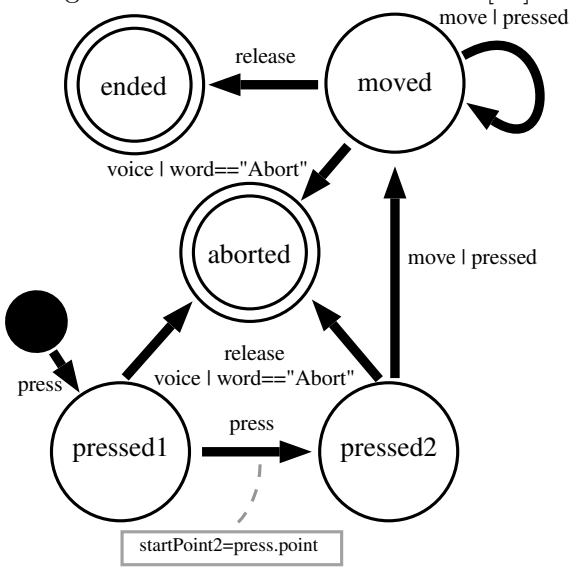


Figure 3.3: Bi-manual interaction [11]

event occurs and the pressed key is ‘a’.

Figure 3.2 illustrates the metamodel of the interaction describing state machine. It is composed of states and transitions. Each transition is between a source state and a target

state. The abstract class *State* represents a state that can be:

- *Initial state:* The first state from where interaction starts.
- *Standard state:* An intermediate state between the initial state and, terminal and aborting states.
- *Terminal State:* The interaction once completed, it comes to this last state. This state cannot be the source of a transition.
- *Aborting state:* If an interaction is aborted, it comes to this state. This state also cannot be the source of a transition like terminal state.

For example, Figure 3.3 illustrates the state machine of bi-manual interaction (this is an instance of the metamodel in Figure 3.2). It starts with the first pressure[3] (*pressed1*). It reaches the state *pressed2* when second pressure occurs. If a *release* event occurs at the states *pressed1* or *pressed2*, the interaction is aborted (*aborted*). If a *moved* event occurs at the state *pressed2*, state *moved* is reached. The interaction is completed on a release event at state *moved*. The interaction can be aborted if the word "Abort" is spoken at the states *pressed1*, *pressed2* or *moved*.

Action

"The life cycle of an action is illustrated in Figure 3.4. It extends the usual action life cycle where actions occur after interactions. Once action is created, it can be executed and updated several times. It can also be aborted while in progress and can also be recycled into another action. After completion of an action if its execution has side effects, it is saved for undo/redo purposes; otherwise the life cycle ends. A saved action can also be removed from the system due to limited size of undo/redo memory." [10]

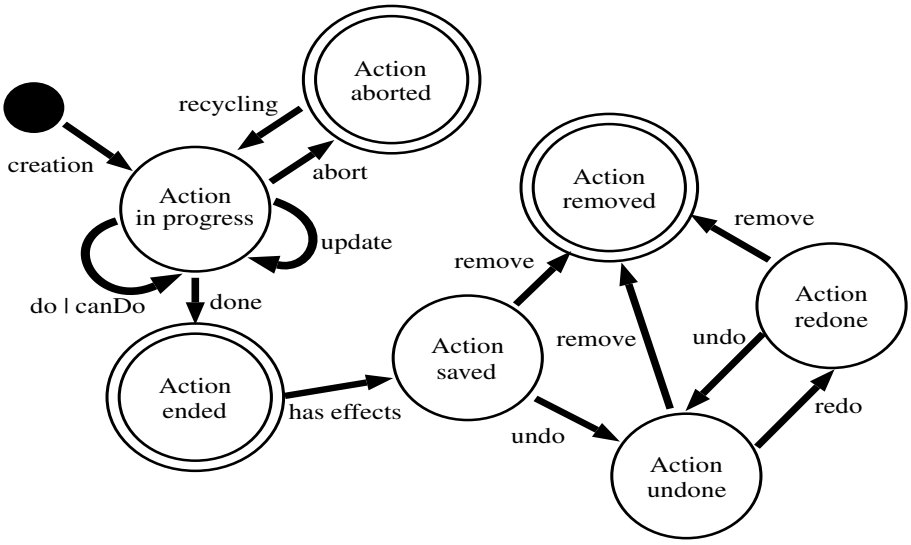


Figure 3.4: Action life cycle [11]

Instrument

In Malai, an instrument is composed of interaction-action links. The metamodel of instru-

[3] A pressure can be produced by a finger, a pencil, or a button of a mouse.

ment is illustrated in Figure 3.5. An interaction-action link is a link between an action that can be executed by the instrument and an interaction that a user carries out to execute actions. Each link is composed of a condition that must be respected to make a link between interaction and action, and an optional interim feedback that provides temporary information related to current interaction.

Figure 3.6 shows pseudo code of an instrument *hand* (this is an instance of the metamodel in Figure 3.5). It describes three interaction-action links. A *simplePress* interaction is used to select shapes (Action *selectShapes*) if the target object is a *shapeUI* (line 3). However, *DnD* interaction (Drag and Drop) is also used to select shapes but if the target object is *CanvasUI* (line 6). This link defines interim feedback (lines 7-9) that provides temporary information related to the selection of shapes.

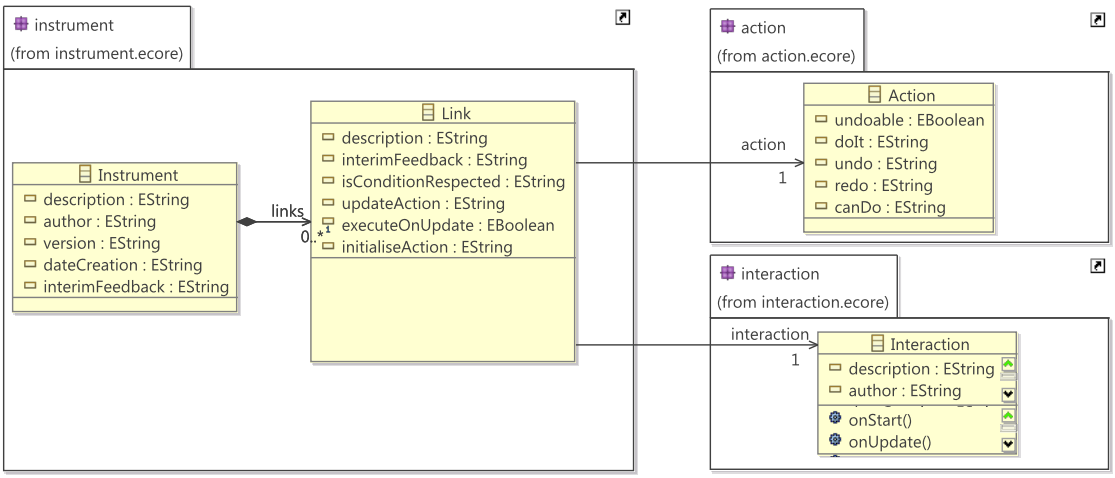


Figure 3.5: Instrument metamodel [10]

```
1 Links Hand hand {
2   SimplePress sp -> SelectShapes action {
3     condition: getObj(sp.point) is ShapeUI
4   }
5   DnD dnd -> SelectShapes action {
6     condition: getObj(dnd.endPt) is CanvasUI
7     feedback : Rectangle rec = new Rectangle(
8                    dnd.startPt, dnd.endPt)
9                hand.setTmpShape(rec)
10  }
11  //...
12  default: hand.setTmpShape(null)
13 }
```

Figure 3.6: Instrument *hand* [11]

3.2.3 Test model

We have discussed the understanding of interactive systems in Section 3.1. We also have defined several test objectives. In order to generate test cases that reveal bugs, we need a model that fulfills our requirements and purposes. Since no models currently are present

that fulfill all requirements and purposes of model-based testers, we extend the models provided by Malai to build the test model that is more suitable to our needs.

The metamodel of the test model is illustrated in Figure 3.7. Extending the models provided by Malai, we can build the test model manually by adding ***succeed*** relation (see *succeed* class) among instruments. In the test model, the *succeed* relation S between two instruments (*i.e.* source I_i, and target I_j instruments) is :

$$S : I_i \preceq I_j$$

This relation means that instrument I_i may occur before I_j, or the activation of I_i eventually activates I_j as well.

Instrument I_j succeeds I_i, if and only if one of the conditions is satisfied.

- The value of the attribute **isInteractionDependable** in *succeed* class is *FALSE* *i.e.* the activation of I_i makes I_j to be activated simultaneously.
- The completion or abortion of an interaction by I_i activates I_j.

As we explained, instruments contain interaction-action links. So in the same way a ***succeed*** relation contains *subLinks*. Each *subLink* refers to a link of I_i (source instrument) and where the true value of the attribute *abortionAllow* signifies that the abortion of the interaction including in the link activates I_j (target instrument). The successful completion of the interaction of the link always activates I_j. Another attribute of *subLink i.e. subCondition* indicates the attributes of the condition of the link that should be satisfied to activate I_j. For example, Listing 3.1 shows pseudo code of the instrument **selectionInstrument** composed of a link between an interaction *SimpleClick* and an action *SelectInstrument*. This instrument helps to select the instruments **Hand** or **Pencil**. Here the activation of the target instrument (*Hand* or *Pencil*) depends on the attributes of the condition (line 3). If the pressed button is *Hand* then it will activate the instrument *Hand*, and if the pressed button is *Eclipse* or *Circle* or *rectangle* then it will activate the instrument *Pencil*.

Listing 3.1: Instruement **SelectionInstrument** pseudo code

```
1 Links SelectionInstrument ins{
2  SimpleClick click -> SelectInstrument action {
3   condition:
4    click.object==buttonHand ||
5    click.object==buttonEclipse ||
6    click.object==buttonCircle ||
7    click.object==buttonRectangle
8  }
9 }
```

In order to find values of attributes of the conditions of interactions' transitions, instruments' links and succeeds' subLinks, we use **choco** [14]. *Choco* is a java library for Constraint Satisfaction Problem (CSP) and Constraint Programming (CP). *choco* is composed of two separate parts: 1) the first part is to express the problem; 2) the second part is actually used to solve the problem. To write a choco program, we need several choco objects:

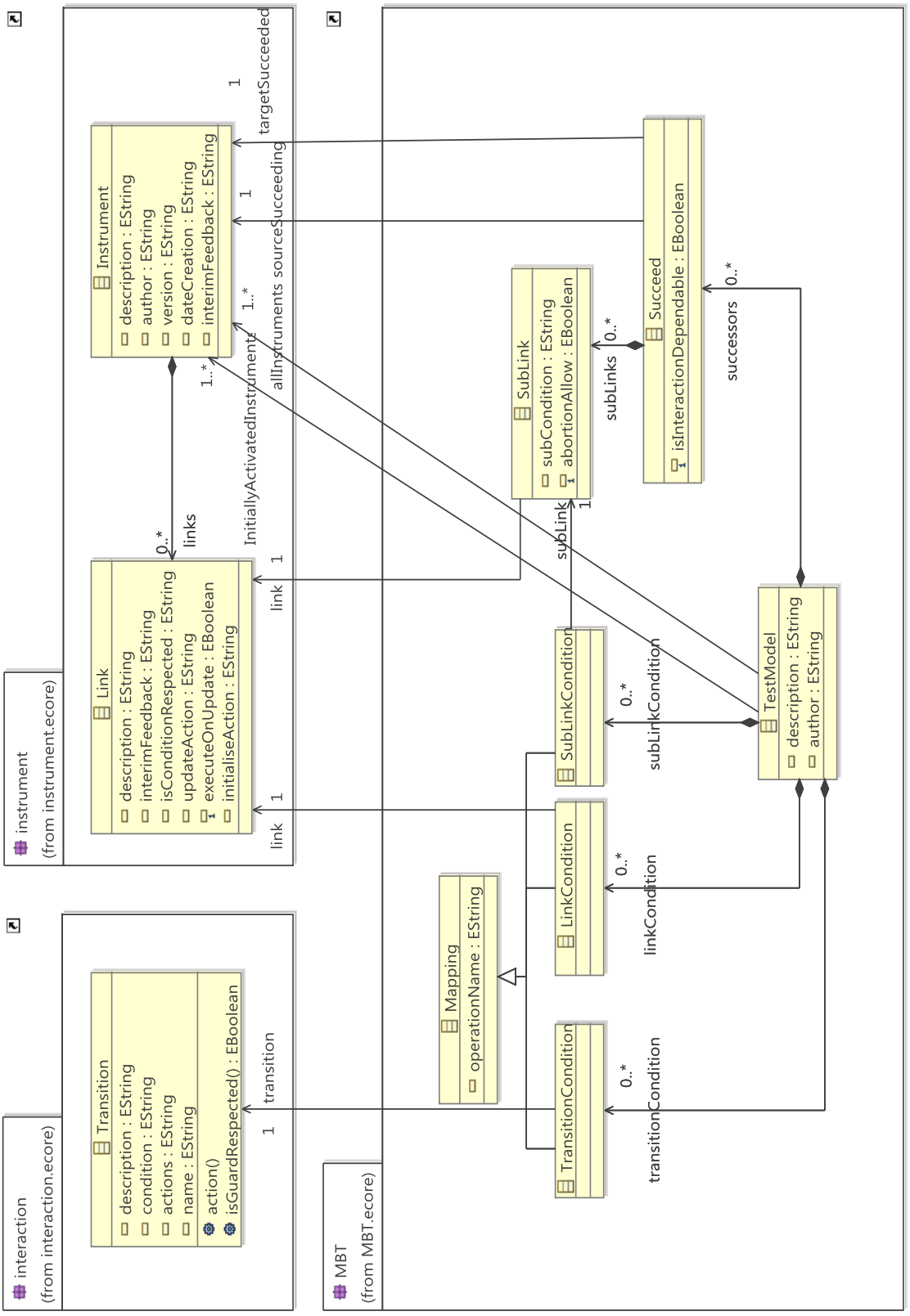


Figure 3.7: Test model metamodel

- **Model**: The model is a central element of a *choco* program that records the variables and constraint defining the problem.
- **Variable**: A variable is defined by a type (*e.g.* integer variable), a name and the values of its domain.
- **Constraint**: A constraint defines relations to be satisfied between variables and constants.
- **Solver**: The solver is used to solver the given problem and helps to read the model.

In the test model, we are modeling conditions in *choco* that will give the values of the attributes of the condition for which the condition is true. The **Mapping** class contains an attribute called *operationName* that is associated to the operation containing the model of the condition. The abstract class *Mapping* can be:

- **Transition condition**: A condition of interactions' transitions.
- **Link condition**: A condition of instruments' links.
- **SubLink condition**: A condition of succeeds' subLinks.

3.2.4 Example

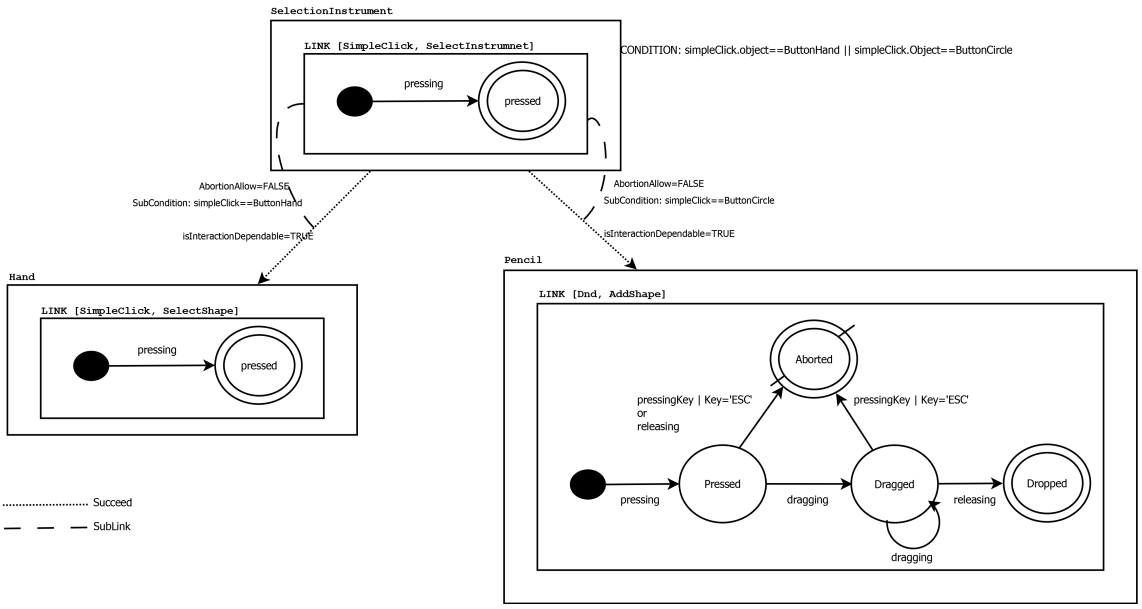


Figure 3.8: Test model example

Figure 3.8 illustrates an example of the test model for three instruments (**selectionInstrument**, **Hand** and **Pencil**). *SelectionInstrument* is used to select either *Hand* or *Pencil* using an interaction **SimpleClick**. So, there are two *succeed* relations existing between *SelectionInstrument* and *Hand*, and *SelectionInstrument* and *Pencil* (*dotted* lines). No instrument is activated simultaneously with the activation of *SelectionInstrument*. So both relations have *TRUE* value for the attribute *isInteractionDependable*. The successful completion of the interaction activates the intended instrument. So, both relations have

subLink relations (*dashed* lines) that refer to the link of *SelectionInstrument*. Since abortion of the interaction does not activate them, they have *FALSE* value for the attribute *abortionAllow*. The link in *selectionInstrument* has a condition that the pressed button should be either hand or circle, but *Hand* is activated if the pressed button is hand and *Pencil* is activated if the pressed button is circle. So, *subLink* relations have *subCondition* of the *SelectionInstrument*'s condition.

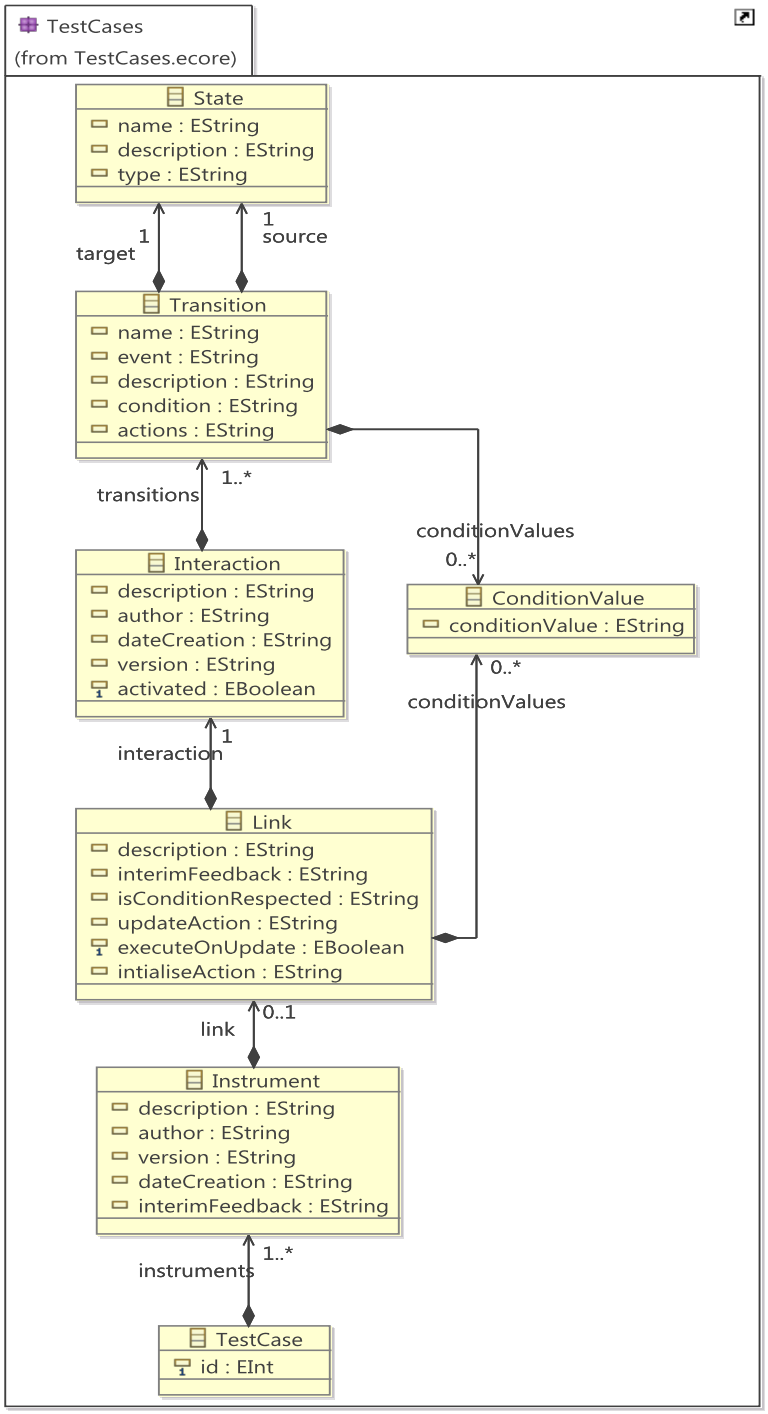


Figure 3.9: Test case metamodel

3.3 Test suite generation

A test suite is a set of test cases. Test cases are generated from the test model explained in the previous section. Our proposed test case metamodel is shown in Figure 3.9. We are considering a sequence of instruments as a test case. Each instrument contains a sequence of an interaction's events of a link. Since links and transitions have conditions in the test model, the links and transitions in a test case also contain the value (obtained by *choco*) for which these conditions are valid.

In order to write algorithms to generate test cases automatically from the test model, we need a language that gives the support to read the test model and transform it (*i.e.* test cases). Since we extending the models of Malai, we are choosing **Kermeta** [1] to write automatic test generation algorithms. Kermeta is a modeling and programming language for meta-model engineering. Kermeta also provides aspect oriented programming and using that we can weave the models to write algorithms. Here, test generation can be seen as a transformation from one instance of the metamodel in Figure 3.7 to a set of test cases instances of metamodel in Figure 3.9. We have implemented three algorithms (or strategies) to generate this set according to 3 coverage criteria (all-path, all-transition, and all-state).

Test coverage criteria play a vital role in model-based testing. They are defined to constitute an adequate test suite (*i.e.* what to test in an interactive system). Test cases are derived from the test model by taking coverage criteria into account. Three different algorithms having different coverage criteria are explained in the following sub-sections. Coverage criteria often relate to one another by *subsumption*. Figure 3.10 illustrates the *subsumption* relations among coverage criteria. All-transition coverage subsumes all-state coverage since if we traverse every transition in a state machine, we will visit every state as well. In the similar way, all-path coverage subsumes all-transition coverage.

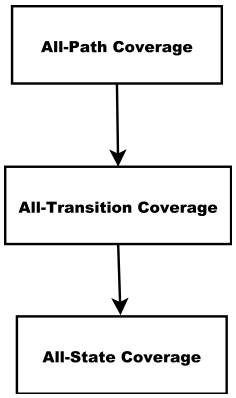


Figure 3.10: Subsumption relations among coverage criteria

3.3.1 All-path coverage criterion

We have explained that a generated test case is a sequence of instruments where each instrument can contain maximum one interaction-action link. Each interaction is represented by a finite state machine. Test cases can be generated by applying some conventional criteria for state machines.

All-path coverage criterion gives all possible paths of instruments from initially activated to the last succeeded instruments. The test case generation algorithm using all-path coverage criterion is shown in Algorithm 1 in Appendix A. A generated test case (TC) is represented as:

$$Inst_1(Link_i(S_i)); Inst_2(Link_j(S_j)); \ldots; Inst_n(Link_k(S_k))$$

Where the first instrument *i.e.* $Inst_1$ in a sequence of instruments is initially activated and $Inst_i(Link_x(S_x))$ signifies a sequence of the interaction associated to a link x of the activated instrument i. But each TC must hold these two conditions:

1. $Inst_i$ succeeds $Inst_{i-1}$, $\forall i \in (2 \ldots n)$
2. $Inst_1 \neq Inst_2 \neq \ldots \neq Inst_n$

Example

Consider the test model shown in Figure 3.8. To make the representation of test case simple, we are assigning a number to each transition. The generated test cases using the Algorithm 1 having *numberLoop*=1 are shown as follows:

[(1),(2)], [(1),(3,7)], [(1),(3,8)], [(1),(3,4,6)], [(1),(3,4,9)], [(1),(3,4,5,6)], [(1),(3,4,5,9)]

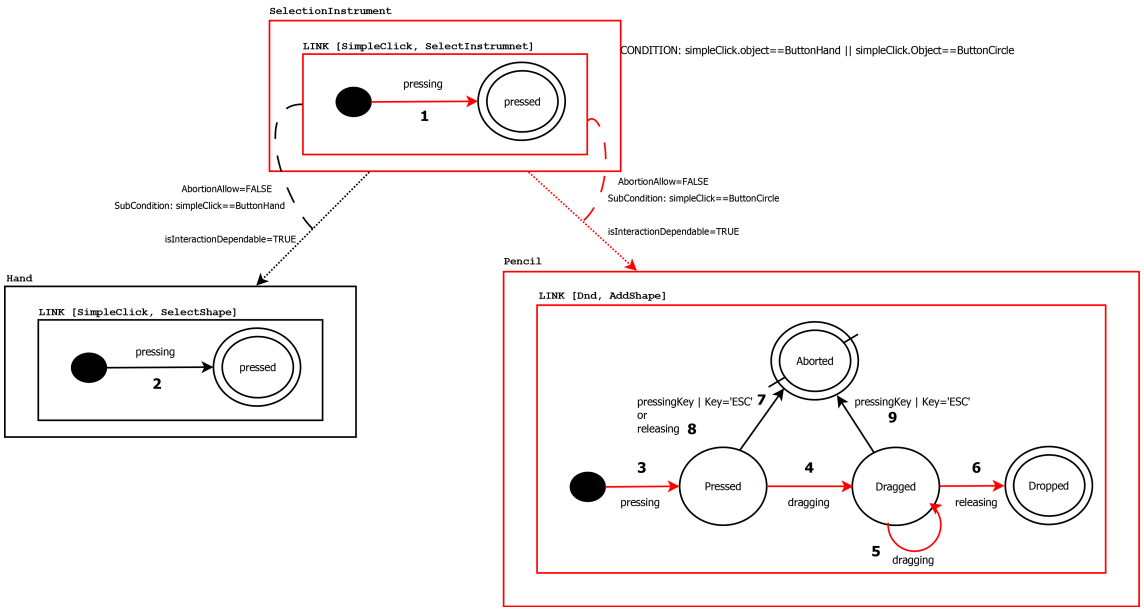


Figure 3.11: All-path coverage criterion example

The test case [(1),(3,4,5,6)] is illustrated in Figure 3.11. The sequence is highlighted by red colour. In the sequence, the *SelectionInstrument* is initially activated instrument. The mouse button is pressed to select the *Circle*. After successful completion of the interaction *SimpleClick*, the *Pencil* is activated. Now a circle shape is created by the interaction *Drag-and-drop*.

3.3.2 All-transition coverage criterion

All-transition coverage criterion gives all paths that visit all transitions of instruments. The test case generation algorithm using all-transition coverage criterion is shown in Algorithm 2 in Appendix A. A generated test case (TC) is similar to a generated TC using all-path coverage criterion:

$$Inst_1(Link_i(S_i)); Inst_2(Link_j(S_j)); \ldots; Inst_n(Link_k(S_k))$$

In this coverage criterion each TC must hold these three conditions:

1. $Inst_i$ succeeds $Inst_{i-1}$, $\forall i \in (2 \ldots n)$
2. $Inst_1 \neq Inst_2 \neq \ldots \neq Inst_n$
3. All transitions visited by current TC $\nsubseteq$ the transitions of an already generated TC

Remark: Number of instruments (n) in a TC may vary if all transitions of the interaction of the succeeded instrument is already visited.

Example

Consider the test model shown in Figure 3.8. The generated test cases using the Algorithm 2 are shown as follows:

[(1),(2)], [(1),(3,7)], [(1),(3,8)], [(1),(3,4,5,6)], [(1),(3,4,5,9)]

As compared to the generated test cases using all-path coverage criterion, [(1),(3,4,6)], [(1),(3,4,9)] are not present since the transitions have already been visited in the test cases [(1),(3,4,5,6)], [(1),(3,4,5,9)].

3.3.3 All-state coverage criterion

All-state coverage criterion gives all paths that visit all states of instruments. The test case generation algorithm using all-state coverage criterion is shown in Algorithm 3 in Appendix A. A generated test case (TC) is similar to a generated TC using all-path coverage criterion:

$$Inst_1(Link_i(S_i)); Inst_2(Link_j(S_j)); \ldots; Inst_n(Link_k(S_k))$$

In this coverage criterion each TC should hold these three conditions:

1. $Inst_i$ succeeds $Inst_{i-1}$, $\forall i \in (2 \ldots n)$
2. $Inst_1 \neq Inst_2 \neq \ldots \neq Inst_n$
3. All states visited by current TC $\nsubseteq$ the states of an already generated TC

Remark: Number of instruments (n) in a TC may vary if all states of the interaction of the succeeded instrument is already visited.

Example

Consider the test model shown in Figure 3.8. The generated test cases using the Algorithm 3 are shown as follows:

[(1),(2)], [(1),(3,7)], [(1),(3,4,6)]

As compared to the generated test cases using all-path coverage criterion, [(1),(3,8)], [(1),(3,4,9)], [(1),(3,4,5,6)], [(1),(3,4,5,9)] are not present since they have already visited all states of all interactions.

We have shown the generated test cases using aforementioned coverage criteria for the test model illustrated in Figure 3.8. Recall that coverage criteria are often related to each other by subsumption. Table 4.1 shows the number of generated test cases. In the table, we can see that the number of generated test cases by all-path coverage is greater than number of all-transition coverage because it subsumes all-transition coverage, and in the similar way, it is between all-transition coverage and all-state coverage.

Coverage criterion	**Number of generated test cases**
All-path	7
All-transition	5
All-state	3

Table 3.1: Number of generated test cases using different coverage criteria

3.4 Executing the test suite

Using our proposed work, all generated test cases are available before executing on the system under test since it is offline-testing (see Section 2.1.1). Once test cases are generated, they can be executed manually or automatically on the system. The generated test cases are abstract test cases. To execute them automatically on the system, they need to be executable. We think that sometimes the automation is not possible. Consider an example of doing bi-manual interaction to zoom in or out an image on iPhone need to be tested manually whereas, simple click interaction to select an instrument on desktop can be done automatically. Having such problems we are executing them manually.

While executing test cases on the system, we need to define test oracle in order to give test verdicts whether they are executed correctly as intended. Each generated test case contains some information to define test oracle such as source and target states of transitions, optional feedback information of links and so forth. As the execution is manual, model-based testers should also have some additional knowledge accumulated from the requirement to define test oracles in their mind.

Consider an example shown in Figure 3.11, a test case **[(1),(3,4,5,6)]** generated using all-path coverage criterion. The sequence of execution is as follow:

1. Instrument *selectionInstrument* is initially activated, check whether it is well selected

2. To launch the *simple click* interaction, click on the button Circle
3. Check whether the instrument *Pencil* is well activated
4. To launch the *drag and drop* interaction, press mouse button in the canvas and verify whether the feedback is shown correctly
5. Drag the mouse and continuously drag it
6. Release the button and check whether a circle has been created.

3.5 Discussion

In this chapter, we have explained our proposed model-based testing approach for post-WIMP interactive systems such as *CPN2000* and *LaTeXDraw*. The test case is built manually by adding the *succeed* relation among instruments. Using three coverage criteria (all-path, all-transition, and all-state coverage), test cases are generated. We believe that using this approach we can test WIMP interactive systems as well since an interaction is composed of low-level events that sometimes associated to widgets.

There are still many aspects that need to be done for the testing of post-WIMP interactive systems. For example, at the moment the generation of test model and the execution of generated test cases are manual. During the execution of test cases manually, sometimes testers cannot be able gather low level details that is needed for an adequate testing. But the details may be gathered if we make the execution of test cases automatic.

In the next chapter, we are applying our testing approach on a post-WIMP interactive system *i.e. LaTeXDraw*.

4

CASE STUDY: LaTeXDraw

Contents

LaTeXDraw is a post-WIMP graphics editor dedicated to the editing of vectorial shapes. LaTeXDraw is based on the Malai architecture. Using the editor, a user is allowed: the creation or deletion of simple shapes (*e.g.* rectangle, ellipse and polygon); the transformation of shapes (*e.g.* re-dimension, rotation, displacement); the editing of shapes (*e.g.* modification of border colour and thickness, displacement of a polygon point).

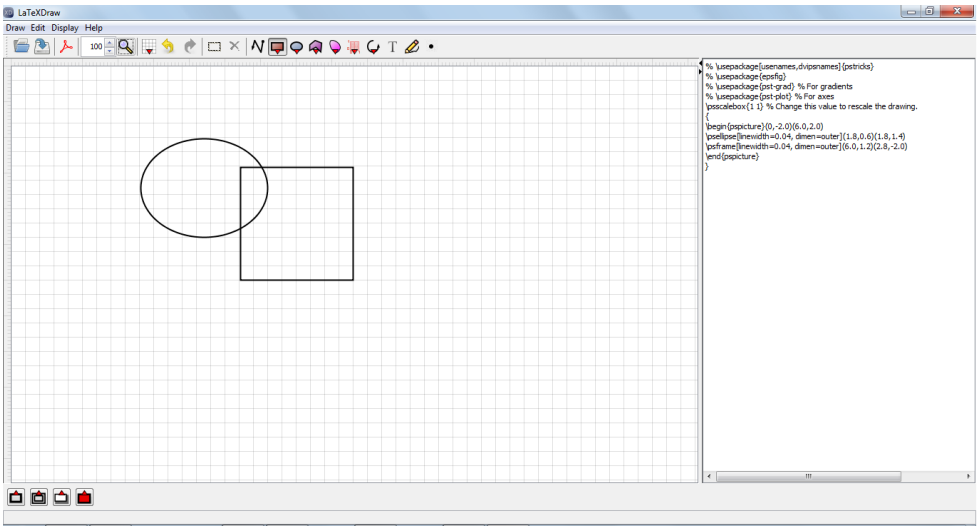


Figure 4.1: LaTeXDraw 3.0

This case study explains the usage of our proposed testing approach by applying it on the LaTeXDraw post-WIMP interactive system. Figure 4.1 shows UI of the LaTeXDraw. For the testing of the system, we are re-using models provided by Malai. These models are described in Section 4.1. The test model described in Section 4.2 is then built manually by adding *succeed* relation among instruments. Test cases are generated automatically from the test model using three different coverage criteria (all-path, all-transition, and all-state coverage criteria). The generated test cases are described in Section 4.3. The manual execution of the generated test cases is described in Section 4.4. The Section 4.5 is concluding the chapter.

4.1 Malai models

In the previous chapter, we have explained several advantages of Malai architecture that lead us to choose it. Malai also provides several models that we are being extended for our testing purpose. Figure 4.2 shows several instruments of the LaTeXDraw 3.0. For the sake of simplicity, we represent each instrument as a class where the name of the class is instrument name and the attributes represent interaction-action links. For example, the class *Hand* is representing the instrument *Hand*. The *Hand* contains two links: 1) the first link *DnD2Select* represents the link between the interaction *drag-and-drop* and the action *Select* used to select shapes included in the area covered by the interaction; 2) the second link *DnD2Translate* represents the link between the interaction *drag-and-drop* and the action *Translate* used to translate the selected shapes. Several interaction models are shown in Figure 4.3. Each interaction model is defined by a finite state machine.

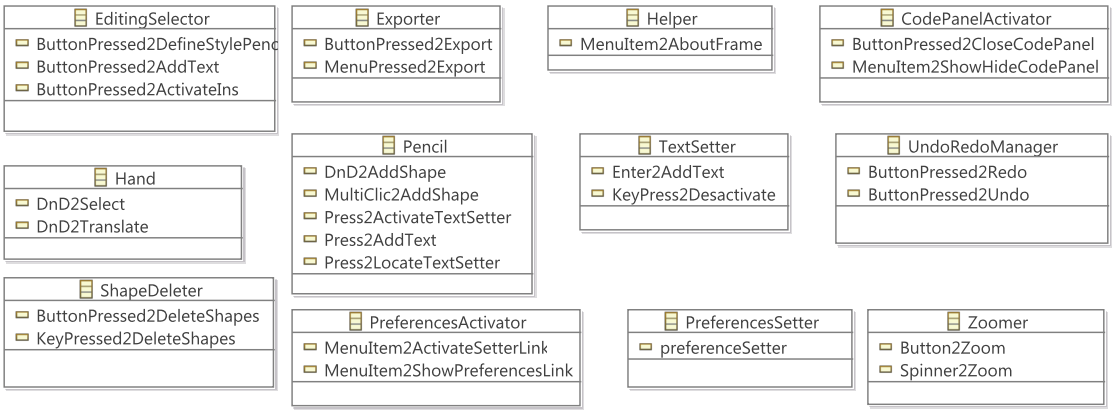
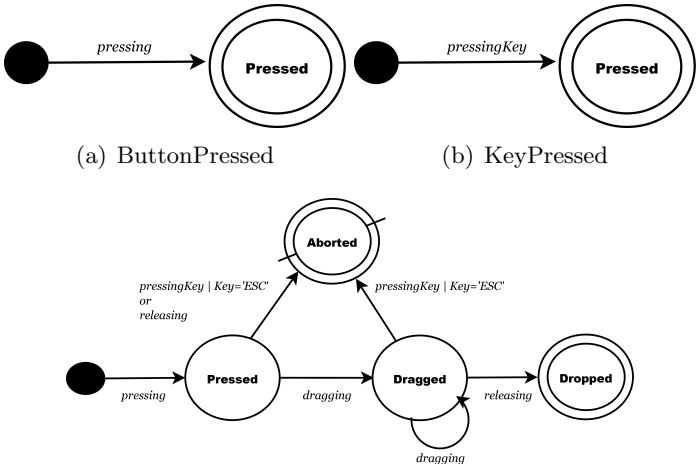


Figure 4.2: Several instruments in LaTeXDraw 3.0

These models are used to build the test model by adding *succeed* relation from which test cases are generated. The test generation of LaTeXDraw is explained in the next section.

4.2 Test model

Once we have models provided by Malai, we can build the test model. The attribute *initiallyActivatedInstruments* holds the instruments which are activated initially when the system starts. In LaTeXDraw, *EditingSelector*, *Exporter*, *Helper*, *CodePanelActivator*,



(a) ButtonPressed (b) KeyPressed

(c) Drag-and-drop

Figure 4.3: Several interaction models

UndoRedoManager, and *Zoomer* are the initially activated instruments. As we explained in the previous chapter, a *succeed* relation between two instruments (I_i and I_j) tells I_j succeed I_i that conforms the two conditions (see Section 3.2.3).

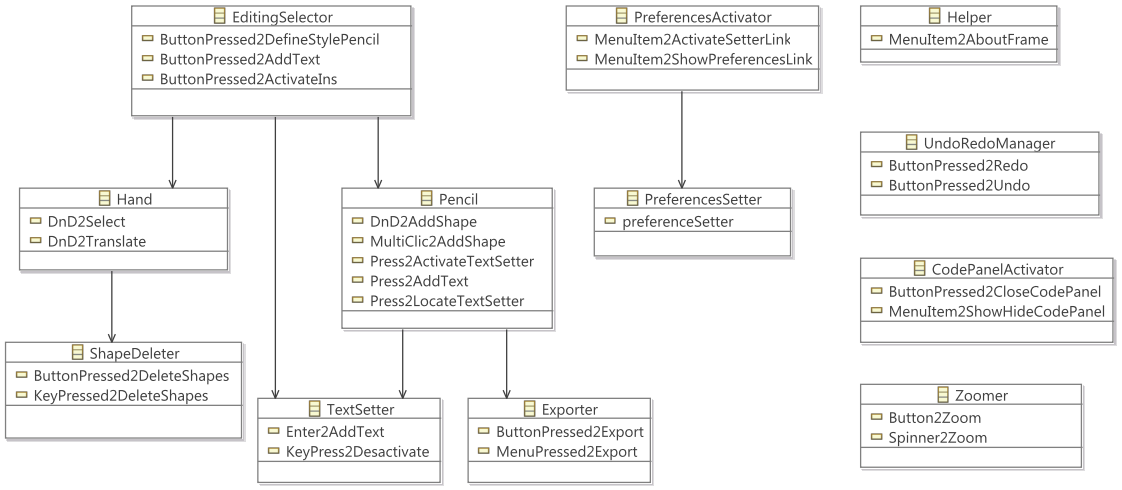


Figure 4.4: Test model for LaTeXDraw 3.0

The test model of the LaTeXDraw is shown in Figure 4.4. The successful completion of the interaction associated to the link *ButtonPressed2AddText* of the *EditingSelector* activates the instrument *TextSetter*. Whereas, the *Hand* and *Pencil* are activated by the successful completion of the interaction provided by the other two links of the *EditingSelector*. The activation of either *Hand* or *Pencil* depends on the condition associated to the links of the *EditingSelector* (*e.g.* if the pressed button is *Rectangle*, the *Pencil* will be activated, and if the pressed button is *Hand*, the *Hand* will be activated). The instrument *ShapeDeleter* succeeds *Hand* since it is activated by the completion or abortion of the interaction associated to the links of the *Hand*. Similarly, the instrument *TextSetter* succeeds *Pencil* since it is activated by the successful completion of the interaction of the

link *Press2ActivateTextSetter* of the *Pencil*. Whereas, the *Exporter* is activated by the abortion or successful completion of the interaction associated to the other links of the *Pencil* (*e.g.* when a shape is created on the canvas, it could be exported as an image). To activate instrument *PreferencceSetter*, the successful completion of the interaction associated to the links of the *PreferencesActivator* must be done. The following section explains how test cases are generated from the test model.

4.3 Test suite generation

A test suite is a set of test cases. Using our proposed approach, a generated test case is a sequence of instruments containing a sequence of an interaction's events of a link. Test cases are generated from the test model using a coverage criterion. We explained three different coverage criteria in Section 3.3. Table 4.1 shows the number of generated test cases of the model shown in Figure 4.4. Recall that coverage criteria are often related to each other by subsumption. In the table, we can see that the number of generated test cases using all-path coverage criterion is greater than all-transition criterion because it subsumes all-transition coverage. In the similar way, the number of generated test cases using all-transition coverage criterion is greater than all-state criterion because it subsumes all-state coverage.

Coverage criterion	**Number of generated test cases**
All-path	142
All-transition	47
All-state	29

Table 4.1: Number of generated test cases for LaTeXDraw 3.0

4.4 Test suite execution

Using our testing approach, once all test cases are generated from the test model, they are executed manually on the system under test. Each test case containing information such as source and target states of an interaction defines a test oracle. While executing test cases manually, the additional knowledge gathered from the specifications and requirements is also used to define test oracles.

When we executed manually the generated test cases on LaTeXDraw, we found several bugs. We have reported these bugs on the website[1] of LaTeXDraw. These bugs are explained in terms of test objectives shown in Section 3.1 as follows:

- In the sequence where *PreferenceActivator* activates *PreferencesSetter*, when we change some values of the preferences, it needs restart of the system to reflect the changes. But no change is reflected *i.e.* the *functional core* data is not updated[2].

[1] `https://launchpad.net/latexdraw`
[2] `https://bugs.launchpad.net/latexdraw/+bug/770726`

- In the sequence where *EditingSelector* activates *Pencil*, when we select the button Bezier curve to draw Bezier curves, the *Pencil* is not activated *i.e. dialog controller* component is not working[3].
- In the sequence where *EditingSelector* activates *Pencil*, when a polygon is drawn, the creation can be aborted by pressing ESCAPE key, but it is not aborted. And In the sequence where *EditingSelector* activates *Hand*, while the translation of the selected shapes is being performed, the translation can be aborted by pressing ESCAPE key, but it is not aborted *i.e.* the *logical interaction* component is not working[4].
- The instrument *Zoomer* is used to zoom in/out the canvas. In the sequence of *Zoomer* interaction, the button to set the zoom value to 100% is not working properly *i.e.* the *logical interaction* component is not working[5].

4.5 Discussion

In this chapter, we applied our proposed model-based testing approach on the post-WIMP interactive system LaTeXDraw. Test cases were generated using three coverage criteria explained in the previous chapter. The generated test cases were then executed manually on the system. We saw that using our approach, we are able to reveal bugs of post-WIMP interactive systems.

[3] `https://bugs.launchpad.net/latexdraw/+bug/788712`
[4] `https://bugs.launchpad.net/latexdraw/+bug/768344`
[5] `https://bugs.launchpad.net/latexdraw/+bug/770723`

CONCLUSION

In this report we introduced a novel model-based approach for testing of interactive systems. Model-based testing approaches use the concept of model built from specifications or requirements of the systems. The model is used to generate test cases that are executed on the systems.

Current testing approaches show their limits in being able to test only WIMP (*Window, Icon, Menu, Pointing device*) interactive systems. The evolution of devices, input devices, output devices, and ways to use these devices (*e.g.* nomadic use) brings changes in the way to create User Interfaces (*UIs*). Post-WIMP interactive systems are now commonly being used (*e.g.* the bi-manual interaction on smart phones). Thus, the testing of these new kinds of interactions and UIs is a challenge to tackle.

We proposed one model-based testing approach for the conventional post-WIMP interactive systems such as *CPN2000*[6] and *LaTeXDraw*[7]. Malai architecture [11] provides several models (*i.e.* instruments, interactions, and actions) that are being extended. A test model is built by adding the *succeed* relation among instruments. A *succeed* relation between two instruments (I_i and I_j) tells I_j succeed I_i if the activation of I_i makes I_j to be activated simultaneously, or the completion or abortion of an interaction by I_i activates I_j. We wrote three algorithms to generate test cases automatically from the test model. The generated test cases are executed manually on the system under test. We demonstrated our proposed testing approach by applying on *LaTeXDraw*. The result shows that the generated test cases from the test model are able to reveal bugs present in the system.

There are still many aspects that need to be done for the testing of post-WIMP interactive systems. Currently, our approach uses manual test model generation and test suite execution. In some cases, manual execution cannot be able to accumulate low level details that is needed for an adequate testing. Whereas, the details may be accumulated if the test suite execution is automatic.

[6] `http://cpntools.org/`
[7] `http://latexdraw.sourceforge.net/`

A

Test case generation algorithm

A.1 Using all-path coverage

The Test case generation algorithm using all-path coverage criterion is shown in Algorithm 1. To make the algorithm simple, we used *visitor*[1] design pattern and we weaved the models using *Kermeta* [23]. Each procedure is represented by **PackageName::ClassName.ProcedureName(*Arguments*)**. To make a call to this algorithm, *testGenerationAllPath* procedure is called. An interaction can have loops, so a variable *numberLoop* (line 4) is declared containing the threshold that controls the looping of the interaction. A temporary instance *i.e. tempTestCase* (line 5) of the test case model is also created that holds the current sequence of instruments from the initially activated instrument to the last succeeded instrument. Once the model is loaded, *visit* procedure of the test model is called (lines 5..6). The *visit* procedure is also called of each initially activated instrument (lines 9..13). If the current activated instrument simultaneously activates other instruments, then they are called first (lines 20..25), else each link is called of the instrument (lines 26..30). An optional condition of link is modeled in *choco* program and the program is called to add the returned result solved by *choco* in the test cases (lines 33..38). Then the *visit* procedure of the interaction of the link is called (line 41). Each interaction is a finite state machine, so its initial state is called first (line 46). The *findTransitions* procedure helps to find the outgoing transitions of a state (lines 47..49) and each outgoing transition is called (line 68) in order to find the sequence, and its target state is also called (line 70). The transition's optional condition is also modeled in *choco* like link conditions. Depending on the class type (*standard*, *terminal* or *aborting*) of states, the *visit* procedure of its target state is called. The succeeded instruments are always called from the terminal state of the current instrument (lines 95..108), but they are called from aborting state if *abortionAllow=TRUE* (line 130). If there is no succeeded instrument, the current value of temporary test case is saved i.e. a **TEST CASE** (lines 90..93, 111..114 and 130..133).

A.2 Using all-transition coverage

The Test case generation algorithm using all-transition coverage criterion is shown in Algorithm 2. This algorithm is similar to the all-path coverage criterion algorithm. To make a call to this algorithm, *testGenerationAllTransition* procedure is called. An interaction can have loops, but it is only visited once so there is no need of a variable *numberLoop*

[1] `http://en.wikipedia.org/wiki/Visitor_pattern`

Algorithm 1: Test Case Generation using All-Path Coverage

```
1  PROCEDURE: testGenerationAllPath(Integer : n)
2  testID ← TestCases::TestID.new
3  testID.ID ← 0
4  numberLoop ← n
5  tempTestCase ← TestCases::TestCase.new
6  model ← load test model
7  model.visit(testID, tempTestCase, numberLoop)
8  PROCEDURE: MBT::TestModel.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, Integer : numberLoop)
9  foreach in ∈ self.initiallyActivatedInstruments do
10 |  outputInstrument ← TestCases::Instrument.new
11 |  tempTestCase.instruments.add (outputInstrument)
12 |  in.visit (testID, tempTestCase, outputInstrument, numberLoop, self)
13 |  tempTestCase.instruments.remove (outputInstrument)
14 PROCEDURE: MBT::TestModel.findSucceededInstruments(Instrument : source)
15 succeededInstruments ← the set of succeeded instruments by the source instrument
16 return succeededInstruments
17 PROCEDURE: instrument::Instrument.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Instrument : outputInstrument, Integer : numberLoop, TestModel :
   mbtModel)
18 outputInstrument ← copy (self)
19 succeededInstruments ← mbtModel.findSucceededInstruments (self)
20 foreach succeed ∈ succeededInstruments do
21 |  if succeed.isInteractionDependable==FALSE then
22 |  |  outputInstrument ← TestCases::Instrument.new
23 |  |  tempTestCase.instruments.add (outputInstrument)
24 |  |  succeed.visit (testID, tempTestCase, outputInstrument, numberLoop, mbtModel)
25 |  |  tempTestCase.instruments.remove (outputInstrument)
26 foreach li ∈ self.links do
27 |  outputLink ← TestCases::Link.new
28 |  outputInstrument.link ← outputLink
29 |  li.visit (testID, tempTestCase, outputLink, outputInstrument, numberLoop, self)
30 |  outputInstrument.link ← null
31 PROCEDURE: instrument::Link.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Link : outputLink, Integer : numberLoop, TestModel : mbtModel,
   Instrument : instrument)
32 outputLink ← copy (self)
33 linkCondition ← find the element that contains the name of the operation (implemented using
   CHOCOframework) which is called and returns values at which this condition is true
34 returnData ← parsing::Conditions.selection (linkCondition.operationName)
35 foreach e ∈ returnData do
36 |  testCond ← TestCases::ConditionValue.new
37 |  testCond.conditionValue ← e
38 |  outputLink.conditionValues.add (testCond)
39 outputInteraction ← TestCases::Interaction.new
40 outputLink.interaction ← outputInteraction
41 self.visit (testID, tempTestCase, outputLink, outputInteraction, numberLoop, mbtModel, instrument, self)
42 outputLink.interaction ← null
43 outputLink.conditionValues.clear()
44 PROCEDURE: interaction::Interaction.visit(TestCases::TestID : testID, TestCases::TestCase
   : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction : outputInteraction,
   Integer : numberLoop, TestModel : mbtModel, Instrument : instrument, Link : link)
45 outputInteraction ← copy (self)
46 self.initState.visit (testID, tempTestCase, outputLink, outputInteraction, numberLoop, self, mbtModel,
   instrument, link)
47 PROCEDURE: interaction::Interaction.findTransitions(State : source)
48 outgoingTransitions ← the set of outgoing transitions of the source state
49 return outgoingTransitions
50 PROCEDURE: interaction::Transition.visit(TestCase::Transition : outputTransition, State
   : source, State : target, TestModel : mbtModel)
51 outputTransition ← copy (self)
52 sourceTransition ← TestCases::State.new
53 sourceTransition ← copy(source)
54 targetTransition ← TestCases::State.new
55 targetTransition ← copy(target)
56 outputTransition.source ← sourceTransition
57 outputTransition.target ← targetTransition
```

```
58 transitionCondition ← find the element that contains the name of the operation (implemented using
   CHOCOframework) which is called and returns values at which this condition is true
59 returnData ← parsing::Conditions.selection (transitionCondition.operationName)
60 foreach e ∈ returnData do
61     testCond ← TestCases::ConditionValue.new
62     testCond.conditionValue ← e
63     outputTransition.conditionValues.add (testCond)
64 PROCEDURE: interaction::InitState.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Link : outputLink, TestCases::Interaction : outputInteraction,
   Integer : numberLoop, Interaction : interaction, TestModel : mbtModel, Instrument :
   instrument, Link : link)
65 outgoingTransitions ← interaction.findTransitions(self)
66 foreach t ∈ outgoingTransitions do
67     outputTransition ← TestCases::Transition.new
68     t.visit(outputTransition, self, t.outputState, mbtModel)
69     outputInteraction.transitions.add (outputTransition)
70     t.outputState.visit(testID, tempTestCase, outputLink, output-
       Interaction, numberLoop, interaction, mbtModel, instrument, link)
71     outputInteraction.transitions.remove (outputTransition)
72 PROCEDURE: interaction::StandardState.visit(TestCases::TestID : testID,
   TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
   outputInteraction, Integer : numberLoop, Interaction : interaction, TestModel : mbtModel,
   Instrument : instrument, Link : link)
73 outgoingTransitions ← interaction.findTransitions(self)
74 foreach t ∈ outgoingTransitions do
75     if t.outputState != self then
76         outputTransition ← TestCases::Transition.new
77         t.visit(outputTransition, self, t.outputState, mbtModel)
78         outputInteraction.transitions.add (outputTransition)
79         t.outputState.visit(testID, tempTestCase, outputLink, output-
           Interaction, numberLoop, interaction, mbtModel, instrument, link)
80         outputInteraction.transitions.remove (outputTransition)
81     else if t.outputState == self and numberLoop != 0 then
82         numberLoop ← numberLoop − 1
83         outputTransition ← TestCases::Transition.new
84         t.visit(outputTransition, self, t.outputState, mbtModel)
85         outputInteraction.transitions.add (outputTransition)
86         t.outputState.visit(testID, tempTestCase, outputLink, output-
           Interaction, numberLoop, interaction, mbtModel, instrument, link)
87         outputInteraction.transitions.remove (outputTransition)
88 PROCEDURE: interaction::TerminalState.visit(TestCases::TestID : testID,
   TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
   outputInteraction, Integer : numberLoop, Interaction : interaction, TestModel : mbtModel,
   Instrument : instrument, Link : link)
89 succeededInstruments ← mbtModel.findSucceededInstruments (instrument)
90 if succeededInstruments == EMPTY then
91     testID.ID ← testID.ID + 1
92     tempTestCase.id ← testID.ID
93     Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
94 else
95     foreach succeed ∈ succeededInstruments do
96         foreach l ∈ succeed.subLinks do
97             if l.link==link then
98                 subLinkCondition ← find the element that contains the name of the operation
                   (implemented using CHOCOframework) which is called and returns values at which
                   this condition is true
99                 outputLink.conditionValues.clear ()
100                returnData ← parsing::Conditions.selection (subLinkCondition.operationName)
101                foreach e ∈ returnData do
102                    testCond ← TestCases::ConditionValue.new
103                    testCond.conditionValue ← e
104                    outputLink.conditionValues.add (testCond)
105                outputInstrument ← TestCases::Instrument.new
106                tempTestCase.instruments.add (outputInstrument)
107                succeed.visit (testID, tempTestCase, outputInstrument, numberLoop, mbtModel)
108                tempTestCase.instruments.remove (outputInstrument)
```

```
109 Hhtbp        PROCEDURE: interaction::AbortingState.visit(TestCases::TestID : testID,
    TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
    outputInteraction, Integer : numberLoop, Interaction : interaction, TestModel : mbtModel,
    Instrument : instrument, Link : link)
110 succeededInstruments ← mbtModel.findSucceededInstruments (instrument)
111 if succeededInstruments == EMPTY then
112 |   testID.ID ← testID.ID + 1
113 |   tempTestCase.id ← testID.ID
114 |   Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
115 else
116 |   foreach succeed ∈ succeededInstruments do
117 |   |   foreach l ∈ succeed.subLinks do
118 |   |   |   if l.link==link and l.abortionAllow then
119 |   |   |   |   subLinkCondition ← find the element that contains the name of the operation
    |   |   |   |   (implemented using CHOCOframework) which is called and returns values at which
    |   |   |   |   this condition is true
120 |   |   |   |   outputLink.conditionValues.clear ()
121 |   |   |   |   returnData ← parsing::Conditions.selection (subLinkCondition.operationName)
122 |   |   |   |   foreach e ∈ returnData do
123 |   |   |   |   |   testCond ← TestCases::ConditionValue.new
124 |   |   |   |   |   testCond.conditionValue ← e
125 |   |   |   |   |   outputLink.conditionValues.add (testCond)
126 |   |   |   |   outputInstrument ← TestCases::Instrument.new
127 |   |   |   |   tempTestCase.instruments.add (outputInstrument)
128 |   |   |   |   succeed.visit (testID, tempTestCase, outputInstrument, numberLoop, mbtModel)
129 |   |   |   |   tempTestCase.instruments.remove (outputInstrument)
130 |   |   |   else if l.link==link and not l.abortionAllow then
131 |   |   |   |   testID.ID ← testID.ID + 1
132 |   |   |   |   tempTestCase.id ← testID.ID
133 |   |   |   |   Save current tempTestCase which represents a sequence of interaction i.e. a TEST
    |   |   |   |   CASE
```

declared in Algorithm 1. A temporary instance *i.e. tempTestCase* (line 5) of the test case model is also created that holds the current sequence of instruments. The main difference between this algorithm and Algorithm 1 is that here each transition is having a variable *visited* that tells whether the transition is already visited (*TRUE*) or not (*FALSE*). If all transitions in the current sequence are not visited, and succeeded instruments are present (not fully visited interaction) then they will be called (lines 111-117 and 132-138). Otherwise, the current value of temporary test case is saved *i.e.* a **TEST CASE** (lines 106..107, 118..120, 127..128 and 139..141).

A.3 Using all-state coverage

The Test case generation algorithm using all-state coverage criterion is shown in Algorithm 3. This algorithm is similar to the all-state coverage criterion algorithm. To make a call to this algorithm, *testGenerationAllState* procedure is called. A temporary instance i.e. *tempTestCase* (line 5) of the test case model is also created that holds the current sequence of instruments. The main difference between this algorithm and Algorithm 2 is that here each state is having a variable *visited* instead of transition that tells whether the state is already visited (*TRUE*) or not (*FALSE*). If all states in the current sequence are not visited, and succeeded instruments are present (not fully visited interaction) then they will be called (lines 101-107 and 122-128). Otherwise, the current value of temporary test case is saved *i.e.* a **TEST CASE** (lines 96..97, 108..110, 117..118 and 129..133).

Algorithm 2: Test Case Generation using All-Transition Coverage

```
1  PROCEDURE: testGenerationAllTransition()
2  Initialize all visited attributes by FALSE
3  testID ← TestCases::TestID.new
4  testID.ID ← 0
5  tempTestCase ← TestCases::TestCase.new
6  model ← load test model
7  model.visit(testID, tempTestCase)
8  PROCEDURE: MBT::TestModel.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase)
9  foreach in ∈ self.initiallyActivatedInstruments do
10 |  outputInstrument ← TestCases::Instrument.new
11 |  tempTestCase.instruments.add (outputInstrument)
12 |  in.visit (testID, tempTestCase, outputInstrument, self)
13 |  tempTestCase.instruments.remove (outputInstrument)
14 PROCEDURE: MBT::TestModel.findSucceededInstruments(Instrument : source)
15 succeededInstruments ← the set of succeeded instruments by the source instrument
16 return succeededInstruments
17 PROCEDURE: instrument::Instrument.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Instrument : outputInstrument, TestModel : mbtModel)
18 outputInstrument ← copy (self)
19 self.visited ← TRUE
20 succeededInstruments ← mbtModel.findSucceededInstruments (self)
21 foreach succeed ∈ succeededInstruments do
22 |  if succeed.isInteractionDependable==FALSE then
23 |  |  outputInstrument ← TestCases::Instrument.new
24 |  |  tempTestCase.instruments.add (outputInstrument)
25 |  |  succeed.visit (testID, tempTestCase, outputInstrument, mbtModel)
26 |  |  tempTestCase.instruments.remove (outputInstrument)
27 foreach li ∈ self.links do
28 |  outputLink ← TestCases::Link.new
29 |  outputInstrument.link ← outputLink
30 |  li.visit (testID, tempTestCase, outputLink, outputInstrument, self)
31 |  outputInstrument.link ← null
32 PROCEDURE: instrument::Link.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Link : outputLink, TestModel : mbtModel, Instrument : instrument)
33 outputLink ← copy (self)
34 linkCondition ← find the element that contains the name of the operation (implemented using
   CHOCOframework) which is called and returns values at which this condition is true
35 returnData ← parsing::Conditions.selection (linkCondition.operationName)
36 foreach e ∈ returnData do
37 |  testCond ← TestCases::ConditionValue.new
38 |  testCond.conditionValue ← e
39 |  outputLink.conditionValues.add (testCond)
40 if self.visited == TRUE then
41 |  testID.ID ← testID.ID + 1
42 |  tempTestCase.id ← testID.ID
43 |  Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
44 else
45 |  foreach t ∈ self.interaction.transitions do
46 |  |  t.visited ← FALSE
47 |  self.visited ← TRUE
48 |  outputInteraction ← TestCases::Interaction.new
49 |  outputLink.interaction ← outputInteraction
50 |  self.visit (testID, tempTestCase, outputLink, outputInteraction, mbtModel, instrument, self)
51 |  outputLink.interaction ← null
52 |  outputLink.conditionValues.clear()
53 PROCEDURE: interaction::Interaction.visit(TestCases::TestID : testID, TestCases::TestCase
   : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction : outputInteraction,
   TestModel : mbtModel, Instrument : instrument, Link : link)
54 outputInteraction ← copy (self)
55 self.initState.visit (testID, tempTestCase, outputLink, outputInteraction, self, mbtModel, instrument,
   link)
56 self.visited ← TRUE
57 PROCEDURE: interaction::Interaction.findTransitions(State : source)
58 outgoingTransitions ← the set of outgoing transitions of the source state
59 return outgoingTransitions
60 PROCEDURE: interaction::Transition.visit(TestCase::Transition : outputTransition, State
   : source, State : target, TestModel : mbtModel)
61 outputTransition ← copy (self)
```

```
62  sourceTransition ← TestCases::State.new
63  sourceTransition ← copy(source)
64  targetTransition ← TestCases::State.new
65  targetTransition ← copy(target)
66  outputTransition.source ← sourceTransition
67  outputTransition.target ← targetTransition
68  transitionCondition ← find the element that contains the name of the operation (implemented using
    CHOCOframework) which is called and returns values at which this condition is true
69  returnData ← parsing::Conditions.selection (transitionCondition.operationName)
70  foreach e ∈ returnData do
71  |   testCond ← TestCases::ConditionValue.new
72  |   testCond.conditionValue ← e
73  |   outputTransition.conditionValues.add (testCond)
74  PROCEDURE: interaction::InitState.visit(TestCases::TestID : testID, TestCases::TestCase :
    tempTestCase, TestCases::Link : outputLink, TestCases::Interaction : outputInteraction,
    Interaction : interaction, TestModel : mbtModel, Instrument : instrument, Link : link)
75  outgoingTransitions ← interaction.findTransitions(self)
76  foreach t ∈ outgoingTransitions do
77  |   outputTransition ← TestCases::Transition.new
78  |   t.visit(outputTransition, self, t.outputState, mbtModel)
79  |   outputInteraction.transitions.add (outputTransition)
80  |   t.outputState.visit(testID, tempTestCase, outputLink, outputInteraction, interaction, mbtModel, instrument, link)
81  |   outputInteraction.transitions.remove (outputTransition)
82  PROCEDURE: interaction::StandardState.visit(TestCases::TestID : testID,
    TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
    outputInteraction, Interaction : interaction, TestModel : mbtModel, Instrument :
    instrument, Link : link)
83  outgoingTransitions ← interaction.findTransitions(self)
84  selfLoop ←Find the transition from outgoingTransitions whose source and target states are same
85  if selfLoop != null then
86  |   selfLoop.visited ←TRUE
87  |   outputTransition ← TestCases::Transition.new
88  |   t.visit(outputTransition, self, t.outputState, mbtModel)
89  |   outputInteraction.transitions.add (outputTransition)
90  |   t.outputState.visit(testID, tempTestCase, outputLink, outputInteraction, interaction, mbtModel, instrument, link)
91  |   outputInteraction.transitions.remove (outputTransition)
92  else
93  |   transitions ←Find the transitions from outgoingTransitions whose source and target states are not
    |   same
94  |   foreach t ∈ transitions do
95  |   |   outputTransition ← TestCases::Transition.new
96  |   |   t.visit(outputTransition, self, t.outputState, mbtModel)
97  |   |   outputInteraction.transitions.add (outputTransition)
98  |   |   t.outputState.visit(testID, tempTestCase, outputLink, outputInteraction, interaction, mbtModel, instrument, link)
99  |   |   outputInteraction.transitions.remove (outputTransition)
100 PROCEDURE: interaction::TerminalState.visit(TestCases::TestID : testID,
    TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
    outputInteraction, Interaction : interaction, TestModel : mbtModel, Instrument :
    instrument, Link : link)
101 succeededInstruments ← mbtModel.findSucceededInstruments (instrument)
102 allVisited ← check that the current interaction's transitions sequence is already visited or not
103 if allVisited == FALSE then
104 |   Make all transitions visited by changing the attribute value to TRUE
105 |   allSucceededLinkVisited ← check that the existing succeeded links are already visited or not
106 |   if succeededInstruments == EMPTY or allSucceededLinkVisited == TRUE then
107 |   |   Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
108 |   else
109 |   |   foreach succeed ∈ succeededInstruments do
110 |   |   |   foreach l ∈ succeed.subLinks do
111 |   |   |   |   if l.terminationVisited == FALSE and l.link == link then
112 |   |   |   |   |   l.terminationVisited ← TRUE
113 |   |   |   |   |   Find the element that contains the name of the operation (implemented using
    |   |   |   |   |   CHOCOframework) which is called and returns values at which this condition is
    |   |   |   |   |   true and add all the returned values to outputLink.conditionValues
114 |   |   |   |   |   outputInstrument ← TestCases::Instrument.new
115 tempTestCase.instruments.add (outputInstrument)
116 succeed.visit (testID, tempTestCase, outputInstrument, mbtModel)
117 tempTestCase.instruments.remove (outputInstrument)
```

```
118 else if l.terminationVisited == TRUE and link == l.link then
119     Find the element that contains the name of the operation (implemented using CHOCOframework)
        which is called and returns values at which this condition is true and add all the returned values to
        outputLink.conditionValues, and save current tempTestCase i.e. a TEST CASE
120 PROCEDURE: interaction::AbortingState.visit(TestCases::TestID : testID,
    TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
    outputInteraction, Interaction : interaction, TestModel : mbtModel, Instrument :
    instrument, Link : link)
121 succeededInstruments ← mbtModel.findSucceededInstruments (instrument)
122 allVisited ← check that the current interaction's transitions sequence is already visited or not
123 if allVisited == FALSE then
124     Make all transitions visited by changing the attribute value to TRUE
125     allSucceededLinkVisited ← check that the existing succeeded links are already visited or not
126     if succeededInstruments == EMPTY or allSucceededLinkVisited == TRUE then
127         Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
128     else
129         foreach succeed ∈ succeededInstruments do
130             foreach l ∈ succeed.subLinks do
131                 if l.abortionAllowVisited == FALSE and l.link == link and
                    l.abortionAllow == True then
132                     l.abortionAllowVisited ← TRUE
133                     Find the element that contains the name of the operation (implemented using
                        CHOCOframework) which is called and returns values at which this condition is
                        true and add all the returned values to outputLink.conditionValues
134                     outputInstrument ← TestCases::Instrument.new
135                     tempTestCase.instruments.add (outputInstrument)
136                     succeed.visit (testID, tempTestCase, outputInstrument, mbtModel)
137                     tempTestCase.instruments.remove (outputInstrument)
138                 else if l.abortionAllowVisited == TRUE and l.link == link and
                    l.abortionAllow == True then
139                     Find the element that contains the name of the operation (implemented using
                        CHOCOframework) which is called and returns values at which this condition is
                        true and add all the returned values to outputLink.conditionValues
140                     Save current tempTestCase which represents a sequence of interaction i.e. a TEST
                        CASE
```

Algorithm 3: Test Case Generation using All-State Coverage

```
1  PROCEDURE: testGenerationAllState()
2  Initialize all visited attributes by FALSE
3  testID ← TestCases::TestID.new
4  testID.ID ← 0
5  tempTestCase ← TestCases::TestCase.new
6  model ← load test model
7  model.visit(testID, tempTestCase)
8  PROCEDURE: MBT::TestModel.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase)
9  foreach in ∈ self.initiallyActivatedInstruments do
10     outputInstrument ← TestCases::Instrument.new
11     tempTestCase.instruments.add (outputInstrument)
12     in.visit (testID, tempTestCase, outputInstrument, self)
13     tempTestCase.instruments.remove (outputInstrument)
14 PROCEDURE: MBT::TestModel.findSucceededInstruments(Instrument : source)
15 succeededInstruments ← the set of succeeded instruments by the source instrument
16 return succeededInstruments
17 PROCEDURE: instrument::Instrument.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Instrument : outputInstrument, TestModel : mbtModel)
18 outputInstrument ← copy (self)
19 succeededInstruments ← mbtModel.findSucceededInstruments (self)
20 foreach succeed ∈ succeededInstruments do
21     if succeed.isInteractionDependable==FALSE then
22         outputInstrument ← TestCases::Instrument.new
23         tempTestCase.instruments.add (outputInstrument)
24         succeed.visit (testID, tempTestCase, outputInstrument, mbtModel)
25         tempTestCase.instruments.remove (outputInstrument)
```

```
26 foreach li ∈ self.links do
27 |   outputLink ← TestCases::Link.new
28 |   outputInstrument.link ← outputLink
29 |   li.visit (testID, tempTestCase, outputLink, outputInstrument, self)
30 |   outputInstrument.link ← null
31 self.visited ← TRUE
32 PROCEDURE: instrument::Link.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Link : outputLink, TestModel : mbtModel, Instrument : instrument)
33 outputLink ← copy (self)
34 linkCondition ← find the element that contains the name of the operation (implemented using
   CHOCOframework) which is called and returns values at which this condition is true
35 returnData ← parsing::Conditions.selection (linkCondition.operationName)
36 foreach e ∈ returnData do
37 |   testCond ← TestCases::ConditionValue.new
38 |   testCond.conditionValue ← e
39 |   outputLink.conditionValues.add (testCond)
40 if self.visited == TRUE then
41 |   testID.ID ← testID.ID + 1
42 |   tempTestCase.id ← testID.ID
43 |   Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
44 else
45 |   Initialize all state's visited attribute to FALSE
46 |   outputInteraction ← TestCases::Interaction.new
47 |   outputLink.interaction ← outputInteraction
48 |   self.visit (testID, tempTestCase, outputLink, outputInteraction, mbtModel, instrument, self)
49 |   outputLink.interaction ← null
50 |   outputLink.conditionValues.clear()
51 |   self.visited ← TRUE
52 PROCEDURE: interaction::Interaction.visit(TestCases::TestID : testID, TestCases::TestCase
   : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction : outputInteraction,
   TestModel : mbtModel, Instrument : instrument, Link : link)
53 outputInteraction ← copy (self)
54 self.initState.visit (testID, tempTestCase, outputLink, outputInteraction, self, mbtModel, instrument,
   link)
55 self.visited ← TRUE
56 PROCEDURE: interaction::Interaction.findTransitions(State : source)
57 outgoingTransitions ← the set of outgoing transitions of the source state
58 return outgoingTransitions
59 PROCEDURE: interaction::Transition.visit(TestCase::Transition : outputTransition, State
   : source, State : target, TestModel : mbtModel)
60 outputTransition ← copy (self)
61 sourceTransition ← TestCases::State.new
62 sourceTransition ← copy(source)
63 targetTransition ← TestCases::State.new
64 targetTransition ← copy(target)
65 outputTransition.source ← sourceTransition
66 outputTransition.target ← targetTransition
67 transitionCondition ← find the element that contains the name of the operation (implemented using
   CHOCOframework) which is called and returns values at which this condition is true
68 returnData ← parsing::Conditions.selection (transitionCondition.operationName)
69 foreach e ∈ returnData do
70 |   testCond ← TestCases::ConditionValue.new
71 |   testCond.conditionValue ← e
72 |   outputTransition.conditionValues.add (testCond)
73 PROCEDURE: interaction::InitState.visit(TestCases::TestID : testID, TestCases::TestCase :
   tempTestCase, TestCases::Link : outputLink, TestCases::Interaction : outputInteraction,
   Interaction : interaction, TestModel : mbtModel, Instrument : instrument, Link : link)
74 outgoingTransitions ← interaction.findTransitions(self)
75 foreach t ∈ outgoingTransitions do
76 |   outputTransition ← TestCases::Transition.new
77 |   t.visit(outputTransition, self, t.outputState, mbtModel)
78 |   outputInteraction.transitions.add (outputTransition)
79 |   t.outputState.visit(testID, tempTestCase, outputLink, outputInteraction, interaction, mbtModel, instrument, link)
80 |   outputInteraction.transitions.remove (outputTransition)
81 PROCEDURE: interaction::StandardState.visit(TestCases::TestID : testID,
   TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
   outputInteraction, Interaction : interaction, TestModel : mbtModel, Instrument :
   instrument, Link : link)
```

```
82  outgoingTransitions ← interaction.findTransitions(self)
83  transitions ←Find the transitions from outgoingTransitions whose source and target states are not same
84  foreach t ∈ transitions do
85      outputTransition ← TestCases::Transition.new
86      t.visit(outputTransition, self, t.outputState, mbtModel)
87      outputInteraction.transitions.add (outputTransition)
88      t.outputState.visit(testID, tempTestCase, outputLink, outputInteraction, interaction, mbtModel, instrument, link)
89      outputInteraction.transitions.remove (outputTransition)
90  PROCEDURE: interaction::TerminalState.visit(TestCases::TestID : testID,
    TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
    outputInteraction, Interaction : interaction, TestModel : mbtModel, Instrument :
    instrument, Link : link)
91  succeededInstruments ← mbtModel.findSucceededInstruments (instrument)
92  allVisited ← check that the current interaction sequence covering states is already visited or not
93  if allVisited == FALSE then
94      Make all states in the sequence visited by changing visited attribute value to TRUE
95      allSucceededInstVisited ← check that the existing succeeded instruments are already visited or not
96      if succeededInstruments == EMPTY or allSucceededInstVisited == TRUE then
97          Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
98      else
99          foreach succeed ∈ succeededInstruments do
100             foreach l ∈ succeed.subLinks do
101                 if l.terminationVisited == FALSE and l.link == link then
102                     l.terminationVisited ← TRUE
103                     Find the element that contains the name of the operation (implemented using
                        CHOCOframework) which is called and returns values at which this condition is
                        true and add all the returned values to outputLink.conditionValues
104                     outputInstrument ← TestCases::Instrument.new
105                     tempTestCase.instruments.add (outputInstrument)
106                     succeed.visit (testID, tempTestCase, outputInstrument, mbtModel)
107                     tempTestCase.instruments.remove (outputInstrument)
108                 else if l.terminationVisited == TRUE and link == l.link then
109                     Find the element that contains the name of the operation (implemented using
                        CHOCOframework) which is called and returns values at which this condition is
                        true and add all the returned values to outputLink.conditionValues
110                     Save current tempTestCase which represents a sequence of interaction i.e. a TEST
                        CASE
111 PROCEDURE: interaction::AbortingState.visit(TestCases::TestID : testID,
    TestCases::TestCase : tempTestCase, TestCases::Link : outputLink, TestCases::Interaction :
    outputInteraction, Interaction : interaction, TestModel : mbtModel, Instrument :
    instrument, Link : link)
112 succeededInstruments ← mbtModel.findSucceededInstruments (instrument)
113 allVisited ← check that the current interaction sequence covering states is already visited or not
114 if allVisited == FALSE then
115     Make all states in the sequence visited by changing visited attribute value to TRUE
116     allSucceededInstVisited ← check that the existing succeeded instruments are already visited or not
117     if succeededInstruments == EMPTY or allSucceededInstVisited == TRUE then
118         Save current tempTestCase which represents a sequence of interaction i.e. a TEST CASE
119     else
120         foreach succeed ∈ succeededInstruments do
121             foreach l ∈ succeed.subLinks do
122                 if l.abortionAllowVisited == FALSE and l.link == link and
                    l.abortionAllow == True then
123                     l.abortionAllowVisited ← TRUE
124                     Find the element that contains the name of the operation (implemented using
                        CHOCOframework) which is called and returns values at which this condition is
                        true and add all the returned values to outputLink.conditionValues
125                     outputInstrument ← TestCases::Instrument.new
126                     tempTestCase.instruments.add (outputInstrument)
127                     succeed.visit (testID, tempTestCase, outputInstrument, mbtModel)
128                     tempTestCase.instruments.remove (outputInstrument)
129                 else if l.abortionAllowVisited == TRUE and l.link == link and
                    l.abortionAllow == True then
130                     Find the element that contains the name of the operation (implemented using
                        CHOCOframework) which is called and returns values at which this condition is
                        true and add all the returned values to outputLink.conditionValues
131                     Save current tempTestCase which represents a sequence of interaction i.e. a TEST
                        CASE
132                 else if l.link == link and l.abortionAllow == FALSE then
133                     Save current tempTestCase which represents a sequence of interaction i.e. a TEST
                        CASE
```

Bibliography

[1] *Kermeta*. `http://www.kermeta.org/`.

[2] *Model-based testing*. `http://en.wikipedia.org/wiki/Model-based_testing`.

[3] *WIMP System*. `http://en.wikipedia.org/wiki/WIMP_(computing)`.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[5] L. Bass, R. Little, R. Pellegrino, S. Reed, R.Seacord, S. Sheppard, and M. Szezur. The arch model: seeheim revisited. *User Interface Developers Workshop*, 1991.

[6] Michel Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-wimp user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, pages 446–453, New York, NY, USA, 2000. ACM.

[7] Michel Beaudouin-Lafon. Designing interaction, not interfaces. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, 2004.

[8] Michel Beaudouin-Lafon and Henry Michael Lassen. The architecture and implementation of cpn2000, a post-wimp graphical application. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, UIST '00, pages 181–190, New York, NY, USA, 2000. ACM.

[9] Cristiano Bertolini and Alexandre Mota. A framework for gui testing based on use case design. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 252–259, Washington, DC, USA, 2010. IEEE Computer Society.

[10] Arnaud Blouin. *Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données*. PhD thesis, Université d'Angers, 2009.

[11] Arnaud Blouin and Olivier Beaudoux. Improving modularity and usability of interactive systems with Malai. In *EICS'10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 115–124, 2010.

[12] Arnaud Blouin, Olivier Beaudoux, and Stéphane Loiseau. Malan: A mapping language for the data manipulation. In *DocEng '08: Proceedings of the 2008 ACM symposium on Document engineering*, pages 66–75. ACM Press, 2008.

[13] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, January 2008.

[14] Choco: an open source Java constraint programming library. `http://sourceforge.net/projects/choco/`.

[15] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.

[16] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] Ibrahim K. El-far. Software testing, analysis, and review conference (starwest 2001), october/november 2001. enjoying the perks of model-based testing.

[18] Ethar Elsaka, Walaa Eldin Moustafa, Bao Nguyen, and Atif M. Memon. Using methods & measures from network analysis for gui testing. In *TESTBEDS 2010: Proceedings of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Alan Hartman. Model based test generation tools overview.

[21] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 245–254, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Antti Kervinen, Mika Maunumaa, and Mika Katara. Model-based testing through a gui. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Software Testing*, volume 3997 of *Lecture Notes in Computer Science*, pages 16–31. Springer Berlin / Heidelberg, 2006. 10.1007/117597442.

[23] Jacques Klein, Jörg Kienzle, Brice Morin, and Jean-Marc Jézéquel. Aspect model unweaving. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 514–530, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] Clemens Nylandsted Klokmose and Michel Beaudouin-Lafon. Vigo: instrumental interaction in multi-surface environments. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 869–878, New York, NY, USA, 2009. ACM.

[25] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1:26–49, August 1988.

[26] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Atif M. Memon. *GUITAR: a GUI Testing frAmewoRk*. `http://guitar.sourceforge.net/`.

[28] Atif M. Memon. *A comprehensive framework for testing graphical user interfaces*. Ph.D., University of Pittsburgh, 2001.

[29] Atif M. Memon. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17:137–157, September 2007.

[30] Atif M. Memon, Student Member, Martha E. Pollack, and Mary Lou Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27:144–155, 2001.

[31] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for guis. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, SIGSOFT '00/FSE-8, pages 30–39, New York, NY, USA, 2000. ACM.

[32] Atif M. Memon and Mary Lou Soffa. Regression testing of guis. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 118–127, New York, NY, USA, 2003. ACM.

[33] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 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*, ESEC/FSE-9, pages 256–267, New York, NY, USA, 2001. ACM.

[34] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, March 2000.

[35] David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.*, 16:18:1–18:56, November 2009.

[36] Duc Hoai Nguyen, Paul Strooper, and Jorn Guy Suess. Model-based testing of multiple gui variants using the gui test generator. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 24–30, New York, NY, USA, 2010. ACM.

[37] Duc Hoai Nguyen, Paul Strooper, and Jörn Guy Süß. Automated functionality testing through guis. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, pages 153–162, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[38] Fabio Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, 1st edition, 1999.

[39] Microsoft Research. *SpecExplorer*. `http://research.microsoft.com/en-us/projects/specexplorer/`.

[40] Hassan Reza, Sandeep Endapally, and Emanuel Grant. A model-based approach for testing gui using hierarchical predicate transition nets. In *Proceedings of the International Conference on Information Technology*, ITNG '07, pages 366–370, Washington, DC, USA, 2007. IEEE Computer Society.

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

[42] Andries van Dam. Post-wimp user interfaces. *Commun. ACM*, 40:63–67, February 1997.

[43] Xun Yuan and Atif M. Memon. Using gui run-time state as feedback to generate test cases. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.

[44] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. Softw. Eng.*, 36:81–95, January 2010.

[45] Lei Zhao and Kai-Yuan Cai. On modeling of gui test profile. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 260–264, Washington, DC, USA, 2010. IEEE Computer Society.