Electronic Communications of the EASST
Volume 22 (2009)


# Proceedings of the Third International Workshop on Formal Methods for Interactive Systems (FMIS 2009)

## UI-Design Driven Model-Based Testing

Judy Bowen and Steve Reeves

16 pages

Guest Editors: Michael Harrison, Mieke Massink
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# UI-Design Driven Model-Based Testing

[1] Judy Bowen and Steve Reeves

[1] University of Waikato, Hamilton, New Zealand

**Abstract:** Testing interactive systems is notoriously difficult. Not only do we need to ensure that the functionality of the developed system is correct with respect to the requirements and specifications, we also need to ensure that the user interface to the system is correct (enables a user to access the functionality correctly) and is usable. These different requirements of interactive system testing are not easily combined within a single testing strategy. We investigate the use of models of interactive systems, which have been derived from design artefacts, as the basis for generating tests for an implemented system. We give a model-based method for testing interactive systems which has low overhead in terms of the models required and which enables testing of UI and system functionality from the perspective of user interaction.

**Keywords:** User interface, prototyping, formal methods, unit testing

## 1 Introduction

Testing of interactive systems is a difficult task. It requires that we test the system's functionality, the interactive behaviour of the system and that the user interface (UI) is usable and aesthetically acceptable for users. As UIs become more complex and software applications become ubiquitous, often relying on new, and sometimes novel, modes of interaction, this difficulty increases. Testing for UIs is often confined to human-based usability testing which is used primarily to ensure users are able to understand and successfully interact with the system under test (SUT) and to measure qualitative responses to aesthetic considerations. While usability testing is an important activity, it is known to be time-consuming and costly and is therefore more successful when performed on systems which have already been well tested.

It is not always practical to separate the testing of system functionality from the UI, and relying on usability testing for interactive elements as well as usability increases time and cost as well as putting a heavier burden on the process in terms of the number and types of errors we rely on it to catch.

In any testing process generating the tests is a critical activity as we want to ensure that the tests have as wide coverage as possible in order to find as many errors as possible, but at the same time we do not want the test generation process to be so onerous that the process becomes impractical due to the length of time it takes and the level of expertise required. In the case of interactive systems the difficulty is again increased as test generation requires knowledge of, and the ability to formally consider, both the underlying functionality and the interactive behaviours.

Model-based testing alleviates some of the problems of test generation in general by providing a formal basis for the tests as well as oracles to compare results against. It lends itself to automatic generation of tests via tool support (see [UL06] for a comprehensive discussion of this) which helps reduce both time and effort required. It also provides a way of generating repeatable

tests and gives confidence in the coverage of the testing. However, model-based testing methods for interactive systems are not yet widespread and have several challenges to overcome if they are to become so.

Choosing an appropriate model to describe the UI is one such issue. Paiva *et al.* [PFV07] for example, highlight and try to address this problem by combining the formal testing framework of a popular programming language, Spec# for the C# language [Spe08], with UML. Their aim is to integrate the formality of Spec# with the visual familiarity of UML to develop abstract models of both functional and UI behavioural requirements which can be used to test for coverage and correctness. They have also chosen state-based models to work with, which is a common choice, but this then requires work to manage the complexity caused by the management of large numbers of states, for example by working with hierarchical models such as those proposed in [PTFV05]. Belli [Bel01, Bel03] extends this idea by using regular expressions to model sequences of user interactions as part of a fault-modelling technique. The exploration of all possible sequences is, however, necessarily large, and the overhead in creating the models not insubstantial.

Comprehensive research on model-based testing for interactive systems has been undertaken by Memon *et al*. (see for example [XM06], [Mem07], [YCM09] and [Mem09]). One of the fundamental concerns of this work is the development of the model to be used for testing. Their methods are based on creating a model of an existing implementation which is then used to develop tests of event and interaction sequences which can be used for regression testing as new functionality is added or the SUT is refactored. In contrast, we are investigating the use of a pre-implementation model of the interactive system which is derived from UI design artefacts and which is linked to a formal specification of the functionality of the system. We aim to find out if such a model can be successfully used to generate tests and provide an oracle to test if a subsequent implementation correctly instantiates the specified interactive system.

In previous work we have developed models for UIs [BR08a, Bow08] which are based upon design artefacts created as part of a user-centred design (UCD) process. In this paper we investigate whether we can use these models as the basis for model-based testing for interactive systems. We propose that this will provide several benefits. Firstly, the models themselves are lightweight and easy to produce as part of standard UCD processes. They use abstraction within the state-based models to avoid state-explosion problems and as such they do not lead to some of the problems associated with other UI models (high overhead of development, complexity of understanding *etc.*) Secondly, we use these models to link the UI and interactive behaviours to a formal system specification which provides a formal model of the entire system enabling us to derive tests which are comprehensive and cover all aspects of the SUT. Thirdly, using the models in this way not only increases the benefits that the use of such models provides, but also enables us to further support UCD techniques formally and test our system from the perspective of interactivity. The models describe both the intended design from the point of view of the UI designer (in conjunction with their informal artefacts such as prototypes) as well as a demonstration of correctness with respect to the overall system and the relationship between system and UI designs. So, we can use the models to derive tests for the properties which have been captured informally and formally within early designs.

Using UI designs as the basis for testing is an approach also taken in [ACE+06] but their work is used as the basis for a test-driven development approach for the UI and follows the approach

of complete separation of UI considerations from underlying functionality. We are concerned with the integration of UI and system behaviours once we are at the point of implementation, and our aim is to use the formal models of the UI to derive tests which ensure correctness of the integration.

The IEEE Software Engineering Body of Knowledge [ISO94] says:

> "Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems."

That is, the purpose of testing is to find errors: a successful test is one that finds an error. In model-based testing the model gives us a description of correct behaviour, so we use this to determine where incorrect behaviour occurs by looking for situations which violate the model. This means finding defects in the functionality and in the way we present that functionality to users, via the user interface. The testing we propose is dynamic, running the program to check behaviour under certain test cases, so it is a post-implementation activity. There are, of course, limitations to model-based testing; we are not guaranteed to find all errors. But by examining the underlying specification for expected behaviours as described in the model we hope to expose as many as possible before we move on to human-based usability testing, which can then focus on finding the sorts of errors which cannot be detected by other testing means.

## 2 Example System

The example we use throughout this paper is a calendar application called SimpleCalendar which is used to display a monthly view of a calendar with events which are assigned to a particular day. The user can view a calendar as a monthly view or as a single day view. They can add events to any given day, view the events of any given day and can also edit or delete those events. Based on these functional requirements a formal specification was developed using Z [ISO02]. Here we give only some of the relevant (to our exposition here) parts of that specification, namely the description of the system state along with descriptions of some of the operations as an example of how the system is specified. We omit, for brevity, the type definitions, axiomatic definitions and the rest of the operations.

The system state contains various observable values: a set *allevents* of events (each event containing a valid date and a title); the current day, month and year; and a set *vdates* which represents the dates currently visible in the application, which is in turn a subset of *allDates*, the set of all valid dates.

The *AddEvent* operation extends the set *allevents* in the *Calendar* state by adding to it the event given in the observation $i?$, the rest of the state remains unchanged. The *RemoveEvent* operation performs the reverse by removing the event given in $i?$ from the set of events.

The *ShowPreviousMonth* and *ShowNextMonth* operations increment or decrement the observation *currentMonth*, and depending on the initial value of *currentMonth* increment or decrement the *currentYear* observation when necessary (if we move forward a month from December or back a month from January).

*Calendar*

$$allevents : \mathbb{P}\, EVENT$$
$$currentMonth : MONTH$$
$$currentYear : \mathbb{N}$$
$$vdates : \mathbb{P}\, allDates$$

*AddEvent*

$$\Delta Calendar$$
$$i? : EVENT$$

---

$$allevents' = allevents \cup \{i?\}$$
$$currentMonth' = currentMonth$$
$$currentYear' = currentYear$$
$$vdates' = vdates$$

*RemoveEvent*

$$\Delta Calendar$$
$$i? : EVENT$$

---

$$allevents' = allevents \setminus \{i?\}$$
$$currentMonth' = currentMonth$$
$$currentYear' = currentYear$$
$$vdates' = vdates$$

*ShowPreviousMonth*

$$\Delta Calendar$$

---

$$allevents' = allevents$$
$$currentMonth > 1 \Rightarrow currentMonth' = currentMonth - 1 \land currentYear' = currentYear$$
$$currentMonth = 1 \Rightarrow currentMonth' = 12 \land currentYear' = currentYear - 1$$
$$vdates' = allDates \rhd (currentMonth' \,..\, currentMonth')$$

*ShowNextMonth*

$$\Delta Calendar$$

---

$$allevents' = allevents$$
$$currentMonth < 12 \Rightarrow currentMonth' = currentMonth + 1 \land currentYear' = currentYear$$
$$currentMonth = 12 \Rightarrow currentMonth' = 1 \land currentYear' = currentYear + 1$$
$$vdates' = allDates \rhd (currentMonth' \,..\, currentMonth')$$

A series of designs and prototypes of the UI for SimpleCalendar were developed following a user-centred design process. At the end of the design iterations the prototypes given in figures 1 and 2 were accepted as the basis for the application's UI.

We create a link between the formal specification of the system and the user interface design by creating presentation models and presentation and interaction models (PIMs) [BR06],

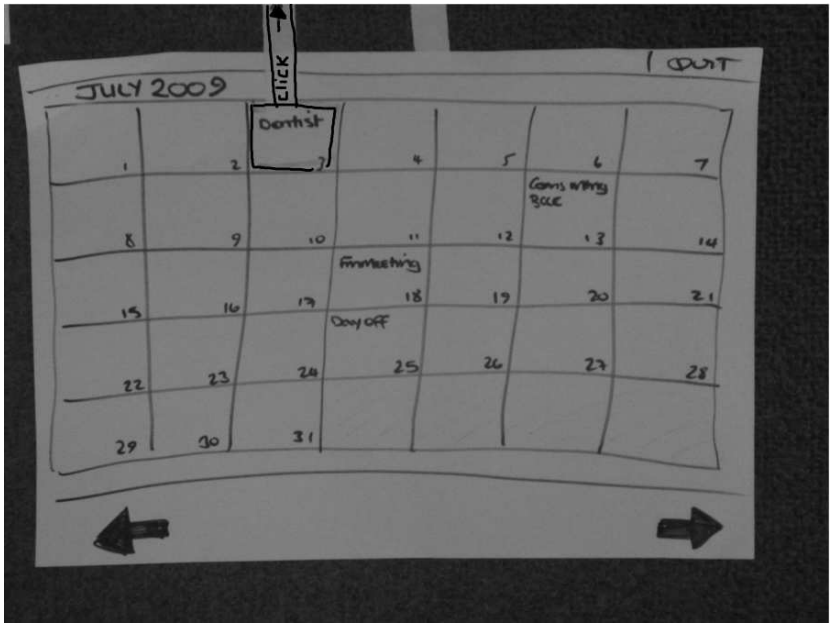


Figure 1: Main Month View for Simple Calendar

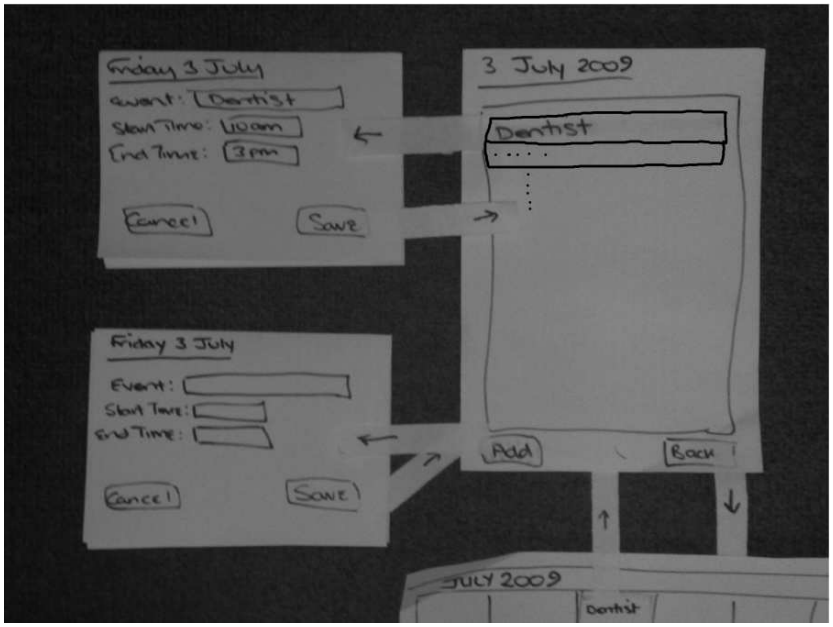


Figure 2: Subsidiary Views for Simple Calendar

[BR08a]. The presentation model gives a description of the interface designs based on the interactive elements (widgets) of the design. Each widget is described by way of a tuple consisting of a name, a category (which determines the type of interactive behaviour it exhibits) and a collection of behaviours associated with the widget. Behaviours either relate to system functionality (*i.e.* provide a way of interacting with the underlying system functionality) or to interface functionality, *e.g.* opening new dialogues, and are prefixed by S_ or I_ respectively. The UI for the entire system is described by a single presentation model which consists of component models for each of the distinct windows and dialogues. For the SimpleCalendar designs this is:

SimpleCal is MainView : DayView : AddView : EditView

MainView is
(QuitButton, ActionControl, (Quit))
(PrevArrow, ActionControl, (S_PrevMonth))
(NextArrow, ActionControl, (S_NextMonth))
(DayDisplay, ActionControl, (I_DayView))

DayView is
(AddButton, ActionControl, (I_AddView))
(EventList, ActionControl, (S_RemoveEvent, I_EditView))
(BackButton, ActionControl, (I_MainView))

AddView is
(TitleEntry, Entry, ())
(StartEntry, Entry, ())
(EndEntry, Entry, ())
(CancelButton, ActionControl, (I_DayView))
(SaveButton, ActionControl, (S_AddEvent, I_DayView))

EditView is
(TitleEntry, Entry, ())
(StartEntry, Entry, ())
(EndEntry, Entry, ())
(CancelButton, ActionControl, (I_DayView))
(SaveButton, ActionControl, (S_UpdateEvent, I_DayView))

We link the UI design models and the specification by creating a presentation model relation (PMR) between each S_Behaviour of the presentation model and operations of the specification, which for our example is *SimpleCalPMR*:

$$\{S\_PrevMonth \mapsto ShowPreviousMonth, S\_NextMonth \mapsto ShowNextMonth,$$
$$S\_RemoveEvent \mapsto DeleteEvent, S\_UpdateEvent \mapsto EditEvent, S\_AddEvent \mapsto AddEvent\}$$

The third model, the PIM, denotes the dynamic behaviour of the UI by describing how each individual dialogue or window is reached by way of I_Behaviours. Each component presentation model is associated with a state of the PIM, and I_Behaviours of the relevant model act as labels

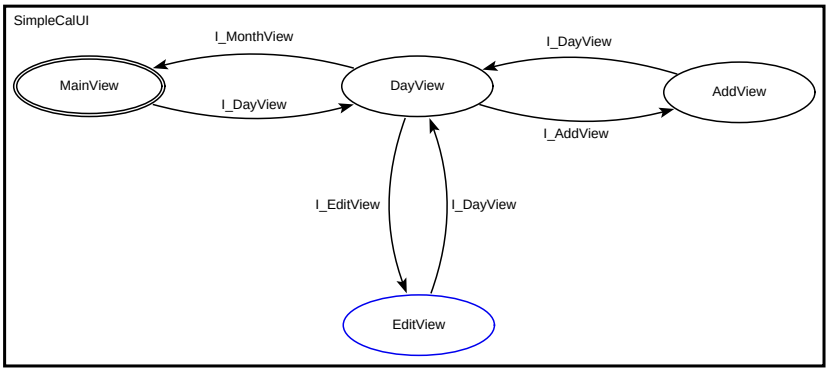


Figure 3: SimpleCal PIM

on transitions between states, and hence, as intended, are behaviours which are purely interface behaviours and so move us around the interface.

The combination of the system specification and the UI models (presentation models, PIM and PMR) provides a formal description of the entire system. We have previously shown how we can use this information as a way of ensuring correctness of the the proposed system [BR08a] and also as the basis for refinement [BR08b]. In this paper, however, we will use the models to derive tests which can then be run on an implementation of the system. The intention is that the models give a description of how we require the implemented system to behave and by using them to generate tests we hope to find errors where the implementation deviates from this behaviour. In the next section we show how the tests are derived.

## 3 Deriving the Tests

The presentation models describe the interactive elements of the UI and their required behaviours. That is, they describe the functionality that is accessible to a user who interacts with the UI. The PIM extends this to describe which behaviours are available in different states of the UI and how a user can move between these states. The testing approach we are proposing will ensure that both the behaviours, and the availability of the behaviours, are provided by the implementation so that we are sure that it satisfies the models. The PIM also describes modality: each independent state of the PIM is modal so we include this as a condition which should be tested.

UI-based testing is often goal-driven. Tasks are defined (or taken from earlier task analysis work) and then sequences of events and user interaction sequences are constructed to satisfy these goals (see for example [Bel01, WA00]). In contrast, the tests we derive use the definitions given within the models as their basis. These tests will be abstract (in that they are expressed at the level of, and in the language of, the models) and can then be instantiated in any language or using any testing framework as required. This will often be dependant on the choice of target implementation language. In section 4 we give an example of one way of instantiating the abstract tests for an implementation of SimpleCalendar in Java.

We begin by considering the dynamic behaviour of the UI. This is defined by I_Behaviours in

the presentation models on transitions of the PIM showing how a user can move between states of the UI. In the PIM given in figure 3 there are four states to be considered, with the initial state being *MainView* (denoted by the double ellipse). For all of the defined behaviours we will test two things: firstly that a widget exists in a given state which provides the required behaviour; and secondly that the behaviour is functionally correct. So, for example, the presentation model for *MainView* describes an *ActionControl* called *DayDisplay* which has a behaviour *I_DayView*. From the PIM we determine that this behaviour should cause the UI to change from the state *MainView* (*i.e.* a state where all of the defined behaviours of *MainView* are available) to the state *DayView* (*i.e.* a state where all of the defined behaviours of *DayView* are available). So first we will test that there is a widget available in *MainView* called *DayDisplay* and then we will ensure that when interaction occurs the UI behaves as required, that is it changes from *MainView* to *DayView*. During the testing process, in order to determine that we are in a correct state, we use the defined behaviours for that state. For example, the state *DayView* is a state of the UI where the behaviours of the *DayView* presentation model are available (a user has access to widgets with the behaviours $I\_AddView, S\_RemoveEvent, I\_EditView$ and $I\_MonthView$). The I_Behaviours and associated widgets for each of the states in our model are:

$$MainView : \{DayDisplay \mapsto I\_DayView\}$$
$$DayView : \{AddButton \mapsto I\_AddView, EventList \mapsto I\_EditView, BackButton \mapsto I\_MonthView\}$$
$$AddView : \{CancelButton \mapsto I\_DayView, SaveButton \mapsto I\_DayView\}$$
$$EditView : \{CancelButton \mapsto I\_DayView, SaveButton \mapsto I\_DayView\}$$

Using this information we derive our first set of tests used to ensure that the relevant widgets exist in the appropriate states. To ensure that a widget is available for a user to interact with we must not only test that it exists in the given state, but also that it is visible and active. We describe the tests using first-order logic (which might be replaced by a table to show which predicates hold for which values in each state if that would be more suitable for various audiences) as follows:

$$\begin{aligned}
&UIState(MainView) \Rightarrow Widget(DayDisplay) \wedge Visible(DayDisplay) \wedge Active(DayDisplay) \\
&\qquad \wedge hasBehaviour(DayDisplay, I_DayView) \\
&UIState(DayView) \Rightarrow Widget(AddButton) \wedge Visible(AddButton) \wedge Active(AddButton) \\
&\qquad \wedge hasBehaviour(AddButton, I_AddView) \\
&UIState(DayView) \Rightarrow Widget(EventList) \wedge Visible(EventList) \wedge Active(EventList) \\
&\qquad \wedge hasBehaviour(EventList, I_EditView) \\
&UIState(DayView) \Rightarrow Widget(BackButton) \wedge Visible(BackButton) \wedge Active(BackButton) \\
&\qquad \wedge hasBehaviour(BackButton, I_MainView) \\
&UIState(AddView) \Rightarrow Widget(CancelButton) \wedge Visible(CancelButton) \wedge Active(CancelButton) \\
&\qquad \wedge hasBehaviour(CancelButton, I_DayView) \\
&UIState(AddView) \Rightarrow Widget(SaveButton) \wedge Visible(SaveButton) \wedge Active(SaveButton) \\
&\qquad \wedge hasBehaviour(SaveButton, I_DayView) \\
&UIState(EditView) \Rightarrow Widget(CancelButton) \wedge Visible(CancelButton) \wedge Active(CancelButton) \\
&\qquad \wedge hasBehaviour(CancelButton, I_DayView) \\
&UIState(EditView) \Rightarrow Widget(SaveButton) \wedge Visible(SaveButton) \wedge Active(SaveButton) \\
&\qquad \wedge hasBehaviour(SaveButton, I_DayView)
\end{aligned}$$

(The predicates here have the obvious (from their names) meaning, for now. They will be given a formal meaning by associating them with computed properties (via pieces of code) later on.) Next we ensure the modality of each state of the PIM (note that it is not necessary to put a

modality requirement on the initial state, *MainView*):

$$UIState(DayView) \Rightarrow Modal(DayView)$$
$$UIState(AddView) \Rightarrow Modal(AddView)$$
$$UIState(EditView) \Rightarrow Modal(EditView)$$

In order to derive tests for the system functionality we similarly identify the widgets with S_Behaviours and ensure that each of the widgets exist and that they have the required behaviours. When we come to instantiate the tests we can use the *PMR* to identify the specified operation which relates to the behaviour and then use the specification to determine the functionality which must be satisfied when the widget is interacted with. The functional tests we derive from the models are, therefore, as follows:

$$UIState(MainView) \Rightarrow Widget(QuitButton) \wedge Visible(QuitButton) \wedge Active(QuitButton) \wedge hasBehaviour(QuitButton, Quit)$$
$$UIState(MainView) \Rightarrow Widget(PrevArrow) \wedge Visible(PrevArrow) \wedge Active(PrevArrow) \wedge hasBehaviour(PrevArrow, S\_PrevMonth)$$
$$UIState(MainView) \Rightarrow Widget(NextArrow) \wedge Visible(NextArrow) \wedge Active(NextArrow) \wedge hasBehaviour(NextArrow, S\_NextMonth)$$
$$UIState(DayView) \Rightarrow Widget(EventList) \wedge Visible(EventList) \wedge Active(EventList) \wedge hasBehaviour(EventList, S\_RemoveEvent)$$
$$UIState(AddView) \Rightarrow Widget(SaveButton) \wedge Visible(SaveButton) \wedge Active(SaveButton) \wedge hasBehaviour(SaveButton, S\_AddEvent)$$
$$UIState(EditView) \Rightarrow Widget(SaveButton) \wedge Visible(SaveButton) \wedge Active(SaveButton) \wedge hasBehaviour(SaveButton, S\_UpdateEvent)$$

Finally we consider the widgets which do not have associated behaviours. In order for our implementation to satisfy the requirements given in the models we must also ensure that these non-functional widgets exist and can be seen by the user. Such widgets are used for a user to provide information to the system by way of inputs or to give information regarding the state of the system back to a user by way of displays.

$$UIState(AddView) \Rightarrow Widget(TitleEntry) \wedge Visible(TitleEntry) \wedge Active(TitleEntry)$$
$$UIState(AddView) \Rightarrow Widget(StartEntry) \wedge Visible(StartEntry) \wedge Active(StartEntry)$$
$$UIState(AddView) \Rightarrow Widget(EndEntry) \wedge Visible(EndEntry) \wedge Active(EndEntry)$$
$$UIState(EditView) \Rightarrow Widget(TitleEntry) \wedge Visible(TitleEntry) \wedge Active(TitleEntry)$$
$$UIState(EditView) \Rightarrow Widget(StartEntry) \wedge Visible(StartEntry) \wedge Active(StartEntry)$$
$$UIState(EditView) \Rightarrow Widget(EndEntry) \wedge Visible(EndEntry) \wedge Active(EndEntry)$$

This is the full set of abstract tests we derive from the models for the SimpleCalendar application. They define all of the conditions on an implementation. The tests provide coverage criteria, we know what we want to test and refer to the fixed properties of the UI which have been given initially within the UI design artefacts (the prototypes of figures 1 and 2 in this example). When we instantiate the tests we will see that we may need to define variables in some instances which are subject to the usual testing considerations of boundaries and choice of values. We discuss this further in the next section and show how we use the current visible state of the UI from a user's perspective to help with these choices.

In the next section we discuss how we instantiated the tests for a Java implementation of SimpleCalendar and give some positive and negative results of the testing process.

# 4 Instantiating and Running the Tests

Having shown how we can derive a set of abstract tests from formal models of UI design artefacts we now give an example of instantiating and running these tests. The Simple Calendar application has been implemented in Java and we have used the FEST testing framework [FES09], which is based on the principles of TestNG and Abbot [RP07], as a way of instantiating and running the tests. While FEST is intended to provide a test-driven development approach to interactive system development, its ability to replicate user interaction (by way of the underlying Java Robot class) makes it a suitable approach for our work. It enables us to take a user-centred approach to our testing in the manner of replicating user interaction with the system to determine correctness of response to possible interaction with the UI and we can then use the underlying support of JUnit to determine whether or not the system behaves as described in our abstract tests. Due to the requirements of FEST classes, which rely on implementation details (such as widget names *etc.*), we take a white-box approach to testing where we use code inspection to determine the information required for FEST (as necessary).

Depending on *how* we want to test the system we might choose different ways of instantiating the tests. For example it may be enough to determine that all required behaviours of all UI states can be accessed by a user, or we might be stricter and require that if our model has two separate controls with a particular behaviour then the tests must show that two such distinct widgets exist with the required behaviour. This is the approach we have taken with this example as it adheres to our commitment to using the designs as the basis for implementation. That is, we expect everything described in the final design artefacts to become part of the implementation.

Just as we did when we began the test derivation process we start by considering the dynamic behaviour of I_Behaviours. In order to determine correctness of state we will ensure that each named state has the correct set of widgets visible to a user and available for interaction. FEST uses a package of classes called *Fixtures* which understand simulation of user events on Java Swing objects and verify the state of these objects. There are different classes for different types of widgets, for example a *JButtonFixture* enables simulation of clicks or double clicks *etc.* upon an actual JButton of an implementation (which is passed to the constructor of the fixture object). In order to test correctness of state, therefore, we create fixtures for each frame or dialogue which instantiates one of the states given in the PIM and then interrogate this to determine whether or not required widgets are present and correctly available. The following code is an example of such a test for the *MainView* state:

```
mv = new FrameFixture(new MView());
public void mViewState(){
  mv.button("quitButton").requireVisible();
  mv.button("quitButton").requireEnabled();
  mv.button("prevArrow").requireVisible();
  mv.button("prevArrow").requireEnabled();
  mv.button("nextArrow").requireVisible();
  mv.button("nextArrow").requireEnabled();
  mv.panel(testdate).requireVisible();
  mv.panel(testdate).requireEnabled();
}
```

where “MView” is the class in our implemented system which provides the UI elements for the *MainView* of the application. When we call the “mViewState()” method from within a JUnit test

method the "MView" frame is created and run in exactly the same way as if we had launched the SimpleCalendar application, and the cursor can be seen moving around the UI over each widget as it identifies it in the same manner as a user moving the mouse to hover over each of the widgets. If any of the tests fail (for example if one of the widgets cannot be found or does not have the required visibility property) we get the standard JUnit red failure bar along with an explanation of the cause of the test failure.

We create similar test methods for *DayView*, *AddView* and *EditView* and then use these as part of our I_Behaviour tests. We can either instantiate each abstract test individually, or combine two or more into a single test. For example we combine the modality requirement given in $UIState(DayView) \Rightarrow Modal(DayView)$ with the state test method for *DayView* by adding "dv.requireModal();" to the state test. In order to instantiate an abstract test such as:

$$UIState(MainView) \Rightarrow Widget(DayDisplay) \wedge Visible(DayDisplay) \\ \wedge Active(DayDisplay) \wedge hasBehaviour(DayDisplay, I\_DayView)$$

we determine from the PIM that a control called *DayView* should have the *I_DayView* behaviour which should change the state of the system from *MainView* to *DayView*. As part of the preparation for our tests we create a FrameFixture called *mv* which allows us to simulate interaction with the UI and take us to any of the other states as required for testing. For example the FEST code for the test given above is:

```
public void mvIDayViewTest(){
  DialogFixture dv = mv.panel(testdate).click().dialog(testdate);
  dViewState(dv);
}
```

This simulates a user clicking on a *dayDisplay* widget (a JPanel in our implementation) which opens a new dialogue, "dv", and we then check that this has the defined *DayView* state. One way of identifying widgets using FEST is by using their name, and in SimpleCalendar we use the current date of each *DayDisplay* panel as the name's value. "Testdate" is a variable containing the current date (as the system always starts up displaying the current month this is a suitable choice for the test variable) and so represents the name of one of the JPanel widgets in *MainView*. This is an example of a test which requires a variable value (a date). Our choice of value for this is made based on what choices are available to a user when the system starts up, so we test based on the dates of the current month and iterate through each of the values that would be visible to the user. The range of the values chosen are then the limits of what a user has access to. We do not randomly test arbitrary dates or seek to test boundary values, such as 01/01/00, 12/12/99 *etc.* as these do not reflect choices the user can make in the current state.

We construct tests as described above for all of the I_Behaviours, and when we run them one at a time we discover our first error. The $dvIAddViewTest()$, which instantiates the abstract test:

$$UIState(DayView) \Rightarrow Widget(AddButton) \wedge Visible(AddButton) \wedge Active(AddButton) \\ \wedge hasBehaviour(AddButton, I\_AddView)$$

fails, producing the error:

```
java.lang.AssertionError: .. property'modal' expected <true> but was <false>
```

When the *aViewState* test is called to ensure that the resulting state after clicking the *Add* button is correct, the modality test fails. In the implementation of SimpleCalendar *AddView* has not been set as a modal dialogue and so the test fails and our error is discovered. Once we have corrected this problem all of the I_Behaviour tests are passed.

We next move onto the non-behavioural widgets, which enables us to test that the implemented UI for SimpleCalendar contains the required widgets for user entry and display. As there are no behaviours associated with these widgets we test them based on their category, so for the abstract test:

$$UIState(AddView) \Rightarrow Widget(TitleEntry) \wedge Visible(TitleEntry) \wedge Active(TitleEntry)$$

We identify the category of *TitleEntry* from the presentation model *Entry* and then instantiate the test by checking that the widget allows user entry (we do not need to test that the widget is visible and active in the state as we have already done this as part of our state tests). Using FEST we simulate the user entering some string into the text field and then test that the value of the text field is the entered string:

```
String tString = "Test Text";
av.textBox("titleEntry").enterText(tString);
av.textBox("titleEntry").requireText(tString);
```

It may seem strange to test the value of the "titleEntry" text box immediately after setting it, but the "enterText" instruction does not set the value of the text box, it merely attempts to interact with it in the same way a user would, by selecting it with the mouse and then entering the keystrokes required to produce the string. If the 'editable' property of the text box was set to false the "enterText" instruction would be carried out (by way of mouse movement and keyboard input) but the textBox would not contain the required string and so the assertion would fail. Each of the non-behavioural widgets are tested in this manner and all of the tests are passed.

Finally we move onto the S_Behaviour widget tests. In order to create these we need to identify and simulate user action on each of the widgets in each state in the same manner as for the I_Behaviours, and use the specified behaviour of operations related via the *PMR* to determine whether or not behaviour is correct. As an example consider the abstract test:

$$\begin{aligned} UIState(MainView) \Rightarrow\ & Widget(PrevArrow) \wedge Visible(PrevArrow) \\ & \wedge Active(PrevArrow) \wedge hasBehaviour(PrevArrow, S\_PrevMonth) \end{aligned}$$

Just as we have done with the other widgets we need to ensure that the widgets are available and visible in the required UI state and that the behaviour is correct. In the case of the S_Behaviours the meaning is given by the specified operation, *ShowPreviousMonth* (described in section 2) which the S_Behaviour is related to via the *PMR*. Because the S_Behaviours enable the user to access the system functionality (and therefore change the system state) as part of our test we should ensure that whenever a user can perform such an operation (*i.e* when a widget with that behaviour is available for interaction) the pre-condition of the related operation holds. This ensures that we do not expose users to the possibility of putting the system into an unexpected state. Secondly we must test that the post-condition given by the invariant in the operation description holds after the interaction, *i.e.* that the correct operation has occurred and has left the system in the expected state. In the example we present in this paper the specification of

the system is given in Z [ISO02] and we use standard conventions for determining pre- and post-conditions for operations. However, it is not a requirement that Z is used, only that related operations can be identified within the given specification and then appropriate methods used to identify the requirements for testing the system state.

For the *PrevArrow* widget in the *MainView* UI state our test then entails the following steps:

- ensure the *PrevArrow* widget exists in the *MainView* state
- ensure the *PrevArrow* widget is visible and enabled in the *MainView* state
- ensure that the pre-condition of the *ShowPreviousMonth* operation holds in *MainView*
- ensure that the post-condition of the *ShowPreviousMonth* operation holds in *MainView* after interaction with the *DayView* widget

The pre-condition of the operation schema can be calculated using standard Z techniques, and can be simplified to $currentMonth = 1 \lor currentMonth \in 2\,..\,12$, which for the UI means testing that the displayed month is either January, or between February and December. The post-condition of the operation requires that we check the value of *allevents* is unchanged and that the visible dates are correctly determined by the new value of *currentMonth* which should be the month prior to the original value. For the FEST testing we are only interested in the UI elements, and therefore separate the non-UI requirements (in this case the condition on *allevents*) into a separate test which can be run using JUnit independently of UI elements. This leads to the following test:

```
public void mvSPrevMonthTest(){
  int cm = cal.get(Calendar.MONTH);
  int year = cal.get(Calendar.YEAR);
  String yearstring = Integer.toString(year);
  mv.label("monthLabel").requireText(makeMonth(cm));
  int pcm = cm -1;
  for(int i = 0; i < 12; i++){
      if(pcm == 11)
          yearstring = Integer.toString(year-1);
      if(pcm == 0)
          pcm = 12;
      String prevdate = makeMonth(pcm);
      mv.button("prevArrow").click();
      mv.label("monthLabel").requireText(prevdate);
      mv.label("yearLabel").requireText(yearstring);
      pcm --;
  }
  sysPostConditionPrevMonth();
}
```

The line “mv.label(“monthLabel”).requireText(makeMonth(cm));” checks the pre-condition by ensuring that the value of the month label is one of the values given by the “makeMonth()” utility method in the test class (which returns only values in the range January to December). The test runs through a twelve month cycle which ensures coverage of both possible post-condition cases irrespective of the start month. Finally the method “sysPostConditionPrevMonth()” is called which is the unit test for the non-UI parts of the post-condition.

As we work our way through the tests for the S_Behaviours we obtain an unexpected result for one of the tests. When we run the test instantiating the abstract test:

$$UIState(DayView) \Rightarrow Widget(EventList) \wedge Visible(EventList) \\ \wedge Active(EventList) \wedge hasBehaviour(EventList, S\_RemoveEvent)$$

We observe the simulated interaction, and conclude that the test should fail. We have created an event titled “Dentist” for a given date, and then test that after S_RemoveEvent this event is no longer displayed in *DayView* or *MonthView*. What we observe upon closure of the *DayView* dialogue is that the event is still displayed in *MonthView*.The test, however, which checks the value of the label displaying event values in *MainView* is passed, as is the JUnit test of the underlying system state which determines that the event has been successfully removed from the collection of events maintained by the system. The error is caused by a lack of graphics refresh by the Java Virtual Machine and so although the event has been correctly removed, and the label text reset to empty, the previous value remains on the screen. Given that it is possible to run all of the tests we created in the background and generate a report of any errors that occur it is quite possible that such an error could be missed by this form of testing. It is a reminder of the importance of performing usability testing with people at the conclusion of model-based testing where such an error would be easily detected.

## 5 Conclusions

In this paper we have shown how our formal models of UI design artefacts can be used as the basis for model-based testing of interactive systems. We showed how it was possible to derive tests and oracles from the models which cover all of the behaviour captured by the UI designs and system specification. The tests are UI driven (as the models are based on UI designs), which reflects our desire to follow a UCD approach supported by formal methods.

We have given an example of how the abstract tests we derive can be instantiated and run against a Java implementation using the FEST framework in conjunction with JUnit. This enabled us to program tests for the implementation (in the nature of white-box testing) and run them to both observe the interaction produced as well as obtain the feedback from FEST and JUnit with respect to whether the tests were passed or not.

During the testing of our example SimpleCalendar application we discovered a modality error where the behaviour of the implementation did not match the oracle given by the model. We also discovered an example of an error which could not be caught by either FEST or JUnit. Our aim in performing model-based testing in this way is to find as many errors as possible prior to performing human-based usability testing. We want to discover as many functional and interaction errors as possible so that user testing can focus on usability and aesthetic issues.

Using the models enabled us to produce a range of abstract tests which covered all of the described interactive behaviours of the UI design models. Further we have shown one way of turning these abstract tests into an implemented test suite that can produce useful results. We believe that this initial investigation into using design models for this purpose has shown it to be a useful area of research to proceed with.

Our tool for creating, editing and storing presentation models and PIMs is currently being extended to support creation and exporting of abstract tests in the manner described in this paper. This will remove the necessity to manually create the abstract tests and may also be able to support partial generation of concrete tests for particular testing strategies. For example we could

automatically generate test method stubs for Java to support the example given in this paper, or use other suitable extensions to the tool depending on how the tests are to be implemented. This seems feasible given the uniform way tests and their predicates are given semantics by code.

We are also interested in investigating this testing strategy further and looking at different ways of instantiating the tests. In particular we would be interested to discover whether alternative methods of instantiation lead to better, or worse, results than we obtained using FEST and JUnit. Given that FEST is intended to be used within a test-driven development (TDD) process we believe it is possible to perform TDD of interactive systems based on the same abstract tests as we have presented here. That is we would use the UI designs as the basis of unit-tests (both for the UI and functionality of the system) and then follow the usual TDD approach of implementing the system with the objective of passing the tests.

Finally we also plan to investigate the use of the the abstract tests presented here as the basis for usability testing. There are many *ad hoc* approaches taken to deciding how a system should be tested with users and we are interested to see if these model-driven tests provide a useful basis for such decisions, and what, if any, differences this leads to in terms of results when compared with task-driven approaches to usability testing.

## Bibliography

[ACE+06] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra. Presenter First: Organizing Complex GUI Applications for Test-Driven Development. *AGILE Conference* 0:276–288, 2006.

[Bel01] F. Belli. Finite-State Testing and Analysis of Graphical User Interfaces. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*. Pp. 34–43. IEEE Computer Society, Washington, DC, USA, 2001.

[Bel03] F. Belli. A Holistic View for Finite-State Modeling and Testing of User Interactions. 2003. Technical Report 2003/1, Institute for Electrical Engineering and Information Technology, The University of Paderborn, April 2003.

[Bow08] J. Bowen. *Formal Models and Refinement for Graphical User Interface Design*. PhD thesis, University of Waikato, Department of Computer Science, 2008.

[BR06] J. Bowen, S. Reeves. Formal Models for Informal GUI Designs. In *1st International Workshop on Formal Methods for Interactive Systems, Macau SAR China, 31 October 2006*. Volume 183, pp. 57–72. Electronic Notes in Theoretical Computer Science, Elsevier, 2006.

[BR08a] J. Bowen, S. Reeves. Formal Models for User Interface design artefacts. *Innovations in Systems and Software Engineering* 4(2):125–141, 2008.

[BR08b] J. Bowen, S. Reeves. Refinement for User Interface Designs. *Electronic Notes Theoretical Computer Science* 208:5–22, 2008.

[FES09] FEST. 2009. FEST (Fixtures for Easy Software Testing). http://fest.easytesting.org/wiki/pmwiki.php

[ISO94] ISO. *ISO/IEC 9646-1—Information Technology—Open Systems Interconnection—Conformance Testing Methodology and Framework, Part 1: General Concepts*. International Standards Organisation. ISO/IEC, first edition, 1994.

[ISO02] ISO. *ISO/IEC 13568— Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*. Prentice-Hall International series in computer science. ISO/IEC, first edition, 2002.

[Mem07] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing Verification and Reliability* 17(3):137–157, 2007.

[Mem09] A. M. Memon. Using Reverse Engineering for Automated Usability Evaluation of GUI-Based Applications. In *Software Engineering Models, Patterns and Architectures for HCI*. Springer-Verlag London Ltd, 2009.

[PFV07] A. Paiva, J. C. P. Faria, R. F. A. M. Vidal. Towards the Integration of Visual and Formal Models for GUI Testing. *Electronic Notes Theoretical Computer Science* 190(2):99–111, 2007.

[PTFV05] A. Paiva, N. Tillmann, J. Faria, R. Vidal. Modeling and testing hierarchical GUIs. In *D. Beauquier, E. Borger, and A. Slissenko, editors, ASM05*. Universite de Paris, 2005.

[RP07] A. Ruiz, Y. W. Price. Test-Driven GUI Development with TestNG and Abbot. *IEEE Software* 24(3):51–57, 2007.

[Spe08] Spec. #. 2008. Microsoft technical pages for Spec #:. http://research.microsoft.com/specsharp/

[UL06] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[WA00] L. White, H. Almezen. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*. P. 110. IEEE Computer Society, Washington, DC, USA, 2000.

[XM06] Q. Xie, A. M. Memon. Model-Based Testing of Community-Driven Open-Source GUI Applications. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Pp. 145–154. IEEE Computer Society, Washington, DC, USA, 2006.

[YCM09] X. Yuan, M. B. Cohen, A. M. Memon. Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*. Pp. 263–266. IEEE Computer Society, Washington, DC, USA, 2009.