A Pattern-Based Approach for GUI Modeling and Testing

Rodrigo M. L. M. Moreira, Ana C. R. Paiva
Department of Informatics Engineering
Faculty of Engineering, University of Porto, Porto, Portugal
Email: {pro08007, apaiva}@fe.up.pt

Atif Memon
Department of Computer Science
University of Maryland, College Park, MD, USA
Email: atif@cs.umd.edu

***Abstract*—User Interface (UI) patterns are used extensively in the design of today's software. UI patterns embody commonly recurring solutions that solve common GUI design problems, such as "login," "file-open," and "search." Yet, testing of GUIs for functional correctness has largely ignored UI patterns. This paper formalizes the notion of a Pattern-Based Graphical User Interface (GUI) Testing method (PBGT) for systematizing and automating the GUI testing process. The space of all possible interactions with a GUI is typically very large. PBGT presents a new methodology to sample the input space using "UI Test Patterns," that embody commonly recurring solutions to test GUIs. Our empirical studies show that the PBGT methodology is effective in revealing faults in fielded GUIs.**

***Index Terms*—GUI Testing, GUI Modeling, Model-based GUI Testing, Software Testing, UI Patterns, UI Test Patterns.**

I. Introduction

Graphical user interfaces (GUIs) appear as fundamental components in today's software. GUIs have become an ideal way of interacting with computer programs, making the software friendlier for its users by offering flexibility in how users perform tasks. For example, while formatting a document in Microsoft's Word software, a user may format a paragraph in a multitude of ways. Users can perform tasks by interacting—via events—with *GUI widgets*; by and large, the sequence of events that they execute are not restricted.

Unfortunately, this flexibility that makes GUIs easy to use also makes testing them for functional correctness notoriously more difficult. The tester has to check whether all—or a reasonable subset—of the possible sequences of events that an end user can execute for a task execute correctly. This task is compounded by the fact that despite all of the advances in automated testing tools and frameworks over the last decade, manual testing still represents the majority of testing effort within most software development organizations. GUI testing in practice remains a huge resource intensive task.

In this paper, we make 2 observations: (1) Developers use specific *tools* to assist the creation of GUIs. The majority of such tools provide a useful contribution to support and enhance teams' productivity in the construction of GUIs. (2) However, these tools and toolkits are only able to support the initial phases of the GUI development process, i.e., they do not provide means to support verification phases.

A specific tool that embodies our 2 above observations is the *user interface (UI) pattern*. UI patterns represent commonly recurring solutions that solve common GUI design problems. Because of the high-level nature of this definition, there are different types of UI patterns at *multiple levels* of abstraction. Perhaps the most widely quoted high-level UI pattern is the Model View Controller (MVC)[1] that advocates a clear division between domain objects that model the real world, and presentation objects that are the GUI elements.

More relevant to our work are more concrete low-level UI patterns that describe specific parts of GUIs, how they should behave on user interaction, and how they should be implemented. For example, the *Double List*[2] pattern that *"is used for making selections. It consists of two sets: all items and the selected items. The user collects a set of items he needs for further processing from one list (or table or tree view etc.) to another. Depending on the goal, the items may be copied or moved to the other list."* or a *Calendar Strip* that *"is a continuous calendar for operating with dates"* and has specific features. Numerous other examples of such concrete patterns abound in text books (e.g., "Object-Oriented Design and Patterns," by Cay Horstmann has a chapter entitled *Patterns and GUI Programming* which discusses "Observer Layout Managers and the Strategy Pattern;" "Containers, and the Composite Pattern"; and "Scroll Bars and the Decorator Pattern") and the web.[3] Despite the abundance of patterns in the GUI design field, model-based testing of GUIs has largely ignored UI patterns.

We posit that GUIs that are similar in design, i.e., based on the same UI pattern, should share a common testing strategy. We develop the notion of a "UI Test Pattern" that provides a configurable test strategy to test a GUI that was implemented using a specific UI pattern or a set of UI patterns. Because a UI pattern may be realized using different implementations, the configurations in a UI test pattern allow us to test different implementations by setting different parameters to the testing strategy. As an example, consider the *Login UI Test Pattern*, discussed in detail later in the paper. This pattern defines a test strategy to test the authentication process that is very common in software applications. However, the authentication process can be implemented differently in different software

[1] martinfowler.com/eaaDev/uiArchs.html
[2] www.cs.helsinki.fi/u/salaakso/patterns/Double-List.html
[3] www.welie.com, time-tripper.com/uipatterns, toastytech.com/guis

applications, e.g., when the authentication fails, a pop-up message may optionally appear. The Login UI Test Pattern may be configured to specify which is the expected behavior in that specific situation and to allow checking if the test passed or failed.

We call our new approach *Pattern Based GUI Testing* (PBGT), which aims to provide a new model-based GUI testing paradigm that promotes reuse of GUI testing strategies. More specifically, the main contributions of PBGT are:

- A testing framework that allows combining test patterns to build newer ones and promote reuse;
- A domain specific language (DSL) – PARADIGM – for GUI modeling;
- A modeling environment (PARADIGM – ME) to support the modeling, test case generation and test case execution;
- A dynamic testing approach that does not need access to the code of the application under test and is cross platform (e.g., web, mobile and desktop).

We empirically evaluate PBGT on two fielded applications: (1) *www.mobile.de*: for buying and selling new and used vehicles, and (2) *australian-charts.com*: a site that shows the best selling music albums in Australia. Our results show that we are able to detect faults in fielded application. We also show that the overall PBGT approach is scalable and usable.

II. Overview

Several software applications have a restricted functionality that is only accessible after a successful authentication. Independently of the specific implementation, a typical authentication mechanism is normally comprised by two input fields – *username* and *password* – and a *submit* button. One implementation of such functionality can be found in *www.mobile.de* and is illustrated in Figure 1.

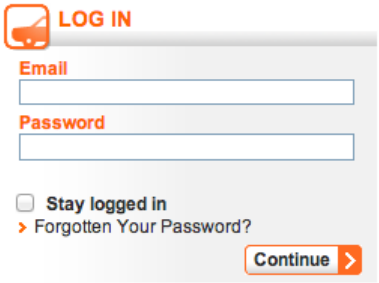

Fig. 1. Mobile.de website Login Form.

In this case, the *username* corresponds to *email* and the *submit* button is labeled as *continue*. Besides those small differences, the overall behavior – provide a username, a password, and submit the data – is the same. Hence, the end-user is able to provide valid and/or invalid inputs: (i) valid username and password; (ii) invalid username and password (because the username is invalid; because the username is left blank or because the password is left blank). In addition, the outcome may vary according to the implementation. For instance, upon authentication failure, the system may display an error message in a particular area of the form or display a pop-up window with the error message. The error message may also vary from "Login failed" to "Invalid username" among others. Concerning a successful log on, we still have different possible implementations, i.e., the end-user may be redirected to a different location or he can stay in the same location but with a different UI that enables higher privileged features that were not available before.

Considering that UI patterns have common behavior that can be implemented in slightly different ways, the goal of PBGT is to provide generic test strategies with possible different configurations in order to allow testing different implementations of the UI pattern promoting reuse. This led to the concept of UI Test Patterns. A UI Test Pattern provides a way to test UI patterns (for instance, authentication) across its several implementations and particular demands.

As such, a UI Test Pattern provides a test strategy with a set of possible tests (or testing goals). Each testing goal has a set of input variables, a sequence of "user" actions, a set of possible checks to perform during test case execution, and a precondition defining the states in which it is possible to perform the actions and the checks. To be executed, the test has to be configured with appropriate test input data for each situation regarding the state of the GUI being tested.

As an example, a UI Test Pattern defining a test strategy for the authentication UI pattern, would have two different testing goals: test the authentication for (1) **valid** and (2) **invalid** Login/Password. The actions to perform will be [fill in *Login*; fill in *Password*; press *Submit*]. In addition, the possible checks to perform could be: "check if it remains in the same page" or "check if a message box pops up" among others. For each testing goal the user/tester defines the input data for the variables Login and Password, selects the checks to perform during test case execution and defines the corresponding precondition. A testing goal can be used several times with different configurations to test a GUI.

Nowadays, UIs are in a constant changing. In this context, the PBGT framework offers the possibility to extend the initial set of UI Test Patterns in order to adapt the test strategies to new trends. This can be done in two different ways: (i) during implementation time, **by the Developer**, extending the framework with additional test patterns; and (ii) during modeling phase, **by the Tester**, defining new UI Test Patterns by combining the existing ones through connectors.

III. UI Test Patterns

Nowadays, it has become noticeable that different GUIs implement common behavior and the majority put into practice the concept of UI patterns providing both similar and recurring solutions (behaviors) for common problems. During this research work, several UI patterns [1]–[6] were identified and analyzed in order to identify general test strategies that could be used to test them. The test solutions defined were named UI Test Patterns (general test solutions for testing common behavior within GUIs).

Definition 1. A UI Test Pattern defines a test strategy which is formally defined by a set of test goals (for later configuration) with the form:

$$< Goal, V, A, C, P >$$

Goal is the ID of the test. *V* is a set of pairs {[variable, inputData]} relating test input data with the variables involved in the test. *A* is the sequence of actions to perform during test case execution. *C* is the set of possible checks to perform during test case execution, for example, "check if it remains in the same page". *P* is a Boolean expression (precondition) defining the set of states in which it is possible to execute the test.

In another way, *Goal* is the "name" of the test goal; *A* and the variables in *V* describe "what" to do and "how" to execute the test. *C* describes the final purpose (or *why*) the test should be executed. *P* defines *when* can be executed.

The *Goal*, variables in *V*, *A* and *C* are defined by the developer during the implementation phase. During the modeling phase, the tester needs to configure each UI Test Pattern within the model. He has to select the test Goals and, for each of those Goals, provide test input data, select the checks to perform, and define the precondition. The tester can select the same test Goal multiple times for a UI Test Pattern providing different configurations. Furthermore, regarding the supply of test input data, UI Test Patterns do not impose any particular technique, so the tester has the freedom to use/select the technique he desires, for instance, *equivalence class partition* (in which the input domain is divided in classes selecting one value from each class, based on the principle that the behavior of the system under test will be the same for every value within the same class) or *boundary value analysis* (select the values that are at the border of equivalence classes, because they are expected to have more probability of finding bugs).

A. Base UI Test Patterns

The UI Test Patterns identified, until now, are denoted as Input, Login, Master/Detail, Find, Sort and Call. We name them Base UI Test Patterns.

Input UI Test Pattern: The Input UI Test Pattern should be used to test the behavior of input fields for valid and invalid input data. Accordingly, this pattern has two possible test Goals: "Valid data" (INP_VD) and "Invalid data" (INP_ID). The set of variables is {input}. The set of possible checks to perform is: {"message box", "label", "error provider"}. The sequence of actions is [provide *input*]. During configuration, the user has to provide valid input data for INP_VD (and invalid input data for INP_ID), select the checks to perform and define the precondition.

Login UI Test Pattern: This test pattern should be used to verify user authentication. The goal is to check if it is possible to authenticate with a valid username/password and check if it is not possible to authenticate otherwise. The Login UI Test Pattern has two possible test goals: "Valid login" (LG_VAL) and "Invalid login" (LG_INV). The set of variables is: {username, password}. The set of checks available is: {"change page", "pop-up error", "same page"}. The sequence of actions (*A*) defined for this test pattern is: [provide *username*; provide *password*; press *submit*]. During configuration the user has to provide valid username/password input data for LG_VAL (and invalid username/password for LG_INV), select the checks to perform and define the precondition.

Master/Detail UI Test Pattern: Master/Detail UI Test Pattern should be applied to GUIs with two related objects (master and detail) in order to verify if changing the master's value correctly updates the contents of the detail. This UI Test Pattern has one possible test goal: "Change master" (MD). The set of variables is {master, detail}. The set of possible checks to perform is {"detail has value *X*", "detail does not have value *X*", "detail is empty"}. The sequence of actions is [select *master*]. During configuration the user has to provide master input data for MD, select the checks to perform and define the precondition.

Find UI Test Pattern: This UI Test Pattern has the purpose to test if the result of a search is as expected (if it finds the right set of values). Moreover, it should be used when someone wants to check the result of a search that shows up after a submit action. The Find UI Test Pattern has two possible test goals: "Value found" (FND_VF) and "Value not found" (FND_NF). The set of variables is $\{v_1,...,v_N\}$ where *N* is defined during configuration time by the user/tester. The set of possible checks to perform is {"empty set", "if it has *X* elements", "if it does not have element *X*", "if the result in line *X* is *Y*"}. The sequence of actions (*A*) is: [provide v_1,... , provide v_N]. During configuration, the user has to define the value for *N*, provide input data for variables v_1,...,v_N, select the check to perform and define the precondition.

Sort UI Test Pattern: The Sort UI Test Pattern is used to check if the result (of a sort action) is ordered accordingly to the chosen sort criterion. The idea is to test user interfaces that contain sortable items/elements, such as, tables and lists. This test pattern has two test goals: "ascending" (SRT_ASC) and "descending" (SRT_DESC). The set of variables is $\{v_1,...,v_N\}$ where *N* is defined by the user/tester during configuration time. The set of possible checks to perform is {"element from field *X* in position *Y* has value *Z*"}. The sequence of actions (*A*) is: [provide v_1,... , provide v_N]. During configuration, the user has to define the value for *N*, provide input data for variables v_1,...,v_N, select the check to perform and define the precondition.

Call UI Test Pattern: The Call UI Test Pattern is used to check the functionality of the corresponding invocation. This test pattern has two test goals: "Action succeeded" (CL_AS) and "Action failed" (CL_AF). This UI Test Pattern does not involve input data so the set of variables is empty. The set of possible checks to perform is {"pop-up message", "stay in the same page", "change to page *X*"}. The sequence of actions (*A*) is: [press]. During configuration, the user has only to select the check to perform and define the precondition.

IV. Implementation

The PBGT framework aims to allow the construction of models where the focus is to model the behavior to test (tester perspective) and not model the behavior of a GUI (user/developer perspective). In order to tailor these objectives, a new domain-specific language (DSL) was developed, hereby called PARADIGM. The DSL was created by taking into account the following method [7]: (i) definition of the PARADIGM's core language model; (ii) definition of the behavior of the PARADIGM language elements; (iii) definition of the PARADIGM's syntax and; (iv) PARADIGM integration with a platform. These activities are described throughout this paper.

A. PARADIGM Language

PARADIGM is a DSL to be used in the domain of PBGT. The goal of the language is to gather applicable domain abstractions (e.g., test patterns), allow specifying relations between them and also provide a way to structure the models in different levels of abstraction to cope with complexity. PARADIGM's meta-model is illustrated in Figure 2.

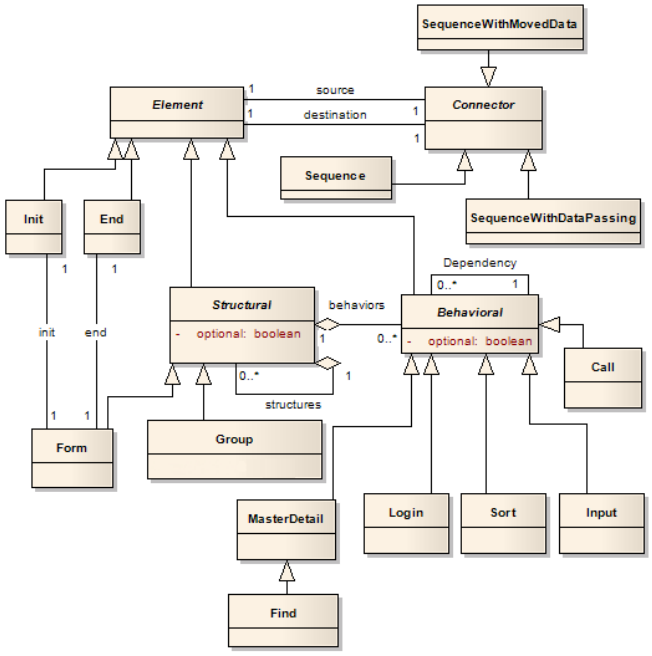

Fig. 2. PARADIGM Language Meta-model.

The PARADIGM language (Figure 2) is comprised by elements and connectors. There are four types of elements: Init (to mark the beginning of a model), End (to mark the termination of a model), Structural (to structure the models in different levels of abstraction), and Behavioral (UI Test Patterns describing the behavior to test).

As models become larger, coping with their growing complexity forces the use of structuring techniques such as different hierarchical levels that allow use one entire model "A" inside another model "B" abstracting the details of "A" when within "B". It is like what happens in programming languages, such as C and Java, with constructs such as modules. Form is a structural element that may be used for that purpose. A Form is a model (or sub-model) with an Init and an End elements. Group is also a structural element but it does not have Init and End and, moreover, all elements inside the Group are executed in an arbitrary order.

The PARADIGM's elements and connectors are described by: (i) an icon/figure to represent the element graphically and; (ii) a short label to name the element. The overall syntax of the DSL is illustrated in Figure 3. Additionally, elements within a model have a number to identify them and, optional elements have a "op" label next to its icon/figure.

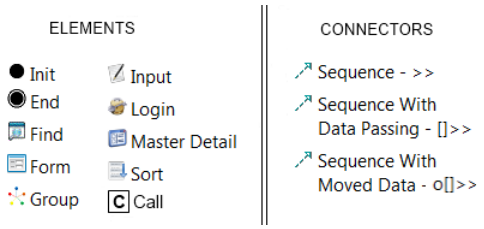

Fig. 3. PARADIGM' Syntax

Connectors: This language has three connectors (the definition of these connectors is based on ConcurTaskTrees - CTT [8]): "Sequence"; "SequenceWithDataPassing"; and "SequenceWithMovedData". The "Sequence (≫)" connector indicates that the target element cannot start until the source element has completed. The "SequenceWithDataPassing ([ ]≫)" connector has the same behavior as "Sequence" and, additionally, indicates that the target element receives data from the source element. "SequenceWithMovedData (o[ ]≫)" has a similar meaning to the "SequenceWithDataPassing" connector, however, the source element transfers data to the target, so the source loses the data that was transferred. In addition, there is another kind of relation among elements - "Dependency" - indicating that the target element depends on the properties of a set of source elements, for instance, when it is the result of a calculation.

1) Constraints: The modeling environment (ME) to support the construction of models written in PARADIGM language (Figure 2) was developed on top of the Eclipse Modeling Framework [9]. Besides imposing the syntax of the language, the ME also imposes language constraints to guaranty building well-formed models. These constraints express restrictions on the way it is possible to dispose elements within a model and how to connect them in order to ensure that GUI models are not ambiguous and allow the subsequent generation of test cases. Some of those constraints are as follows:

- A Connector cannot connect an element to itself;
- A Connector cannot have Init as destination neither End as source;
- An Init element cannot connect directly to an End element;
- Two elements cannot be connected more than once by

connectors of the same type;
- Two Elements can only be connected if they belong to the same Structural Element (Model; Form; Group);
- Elements inside a Form (but not inside Groups of that Form) cannot be loose, i.e., for all elements within a Form, there is at least one path from the Init to the End that traverses that element.

B. High-order UI Test Patterns

In order to promote, even further, reuse when constructing GUI models for testing, PARADIGM language can be extended. It allows the definition of additional test patterns (behavioral elements) on top of the existing base patterns (Input, Login, Master/Detail, Find, Sort and Call). The newer test patterns are called Higher Order Patterns (HOP).

Besides allowing to structure the models in different levels of abstraction, Form element can be seen has an extension mechanism of the language that supports the definition of Higher Order Patterns (HOP). HOP can be created by Testers and are always comprised by at least two other (either BP or HOP) patterns combined by the existing connectors.

The main benefit of the language extension possibility, is that each Tester may have his own set of UI Test Patterns that form a kind of test library that can evolve as desired according to the specific characteristics of the GUIs they test. Consider a HOP (*h*) resulting from the combination between two BP (b_1 and b_2): the set of tests of *h* will be the union between the tests of b_1 and b_2.

C. Test Case Generation

After a model has been constructed and configured, PARADIGM ME is able to generate automatically test cases from such model. Test cases are generated according to a set of rules. Generally, the algorithm traverses the model elements considering the connectors along the way. During test execution, whenever a behavioral element is traversed, the set of the checks, previously defined (configured) by the tester, are performed. In addition, elements within the model can be optional. When an element is optional, it means that there will be test cases that skip the corresponding test strategy. Group elements have a particularity: the inner elements can be executed arbitrarily. Regarding test case generation, it means that there will be test cases for different permutations of those elements. Regarding this particularity, the maximum number of test paths for Groups is given by the following formulas.

Definition 2. A Group with *N* loose mandatory elements, will generate at most

$$N!$$

test paths.

Definition 3. A Group with *M* loose optional elements, will generate at most

$$\sum_{i=M-1}^{M} i! \cdot {}^{M}C_i + 1$$

test paths.

Definition 4. A Group with *MT* mandatory and *OP* optional loose elements will generate at most

$$MT! + \sum_{i=OP-1}^{OP} (MT+i)! \cdot {}^{OP}C_i$$

test paths.

Definition 5. Two connected Groups g_1 and g_2 with t_1 and t_2 number of test paths each, will generate at most (if precondition of internal elements is True)

$$t_1 \cdot t_2$$

test paths.

Definition 6. Overall, a set of *M* test paths, where some of them (*N*) have a Group with *T* number of test paths will generate at most

$$(M-N) + N \cdot T$$

test paths.

Consider for all permutations of elements within groups and further permutations for samples with different lengths when that group has optional elements gives rise to many test paths. For example, in the case of N optional elements within a Group, we could generate test cases for samples with 2 elements, 3 elements, .. , N-1 elements. In the formulas above we have only considered permutations over samples of N-1 elements. Yet it is easy to check that, even for a small model and in case of preconditions true, the number of test cases generated is enormous.

Despite providing an algorithm for test case generation, the framework allows the addition of new algorithms for that purpose during implementation time, by the developer.

V. Demonstration

The objective of this section is to illustrate the PBGT approach and evaluate whether it is possible to use PARADIGM language for modeling real applications available on the web and to use the overall approach for finding errors.

The subject for this demonstration section is a German website (*www.mobile.de*) for buying and selling new and used vehicles [10]. In addition to the features available to the public: search vehicles according to various parameters, sort the search result according to a set of parameters, refine the search, among others, the site also allows registration for users who, after authentication, can insert vehicles for sale, among other functionalities. For illustration purposes and due to space limitations, the model of *Mobile.de* presented in the sequel does not contain all the testing goals needed to test the overall functionality of the site. However, unless the Call UI Test Pattern, the models constructed for the selected testing goals illustrate the use of all the other Base UI Test Patterns of the language PARADIGM.

A. Mobile.de Model

Mobile.de has a page (Car Search) where it is possible to set the search parameters and see the number of elements in the search result set. Upon submission, another page (Show results) shows the elements within the results set obtained. Generally, the goal is to test whether the search functionality works correctly, i.e., if it is possible to search for models of a particular car brand, check whether the number of the cars within the results set is correct and has the elements is supposed to have, check if it possible to sort the results providing different criteria, check if it is possible to refine the search and, in particular, check if the items selected to limit the search are transferred to the search criteria area and vice versa.

The overall model of the *Mobile.de* is structured in different levels of abstraction. Figure 4 corresponds to the first level of the model and presents an overview of the site functionality to test. The two pages of the site are modeled by *Form*s (CarSearch and ShowResults) and the flow of data between those forms is modeled via the "Sequence with passing data" connector. Subsequent levels of the model will detail the inner testing goals of the forms.

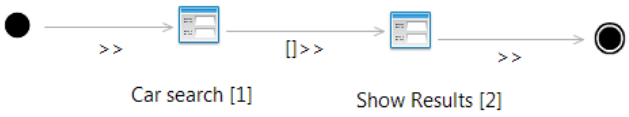

Fig. 4. Mobile.de website Model

The CarSearch *Form* is detailed in Figure 6. It contains one *Group* element and a *Find* UI Test Pattern. The *Group* element includes three *MasterDetail* UI Test Patterns to model the relation between the brand of a vehicle (the "Make" combo box in Figure 5) and the models available for that brand ("Model" combo box in Figure 5) and one *Input* UI Test Pattern (referred in Figure 6 as ChangeLanguage) to change the language of the site (e.g., English, Spanish, etc.). This *Group* is optional which means that the corresponding test strategy of the inner elements can be globally skipped. The subsequent *Find* UI Test Pattern gets data from the *Group* (modeled by the connector "[ ]≫") and submits it to the following element.

The CarSearch *Form* is connected to the ShowResults *Form* through the "[ ]≫" connector (Figure 4). The Show Results *Form* models the page of the site that shows the results of the search configured previously (Figure 7). In addition, this

Fig. 5. Mobile.de Car Search Form

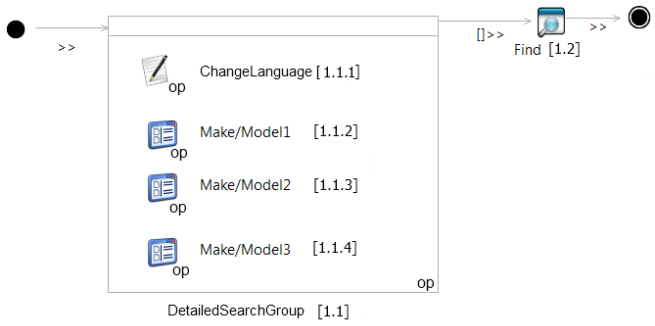

Fig. 6. Mobile.de Car Search Form Model

page supports refining the search (through adjust and limit search); navigating throughout the result set; and changing the overall system's language among other functionalities. The Show Results *Form* is detailed in Figure 8. It is comprised by an *Init*, a *Group* element (ResultsGroup) and an *End*. The *Group* (ResultsGroup) is optional and is comprised by another *Group* (referred as FilterGroup), a *Sort*, a *Find* and an *Input* UI Test Patterns. The ResultsGroup receives data from the *Init* element ("[ ]≫") (in this case it means that it gets data from the CarSearch *Form* within Figure 4.

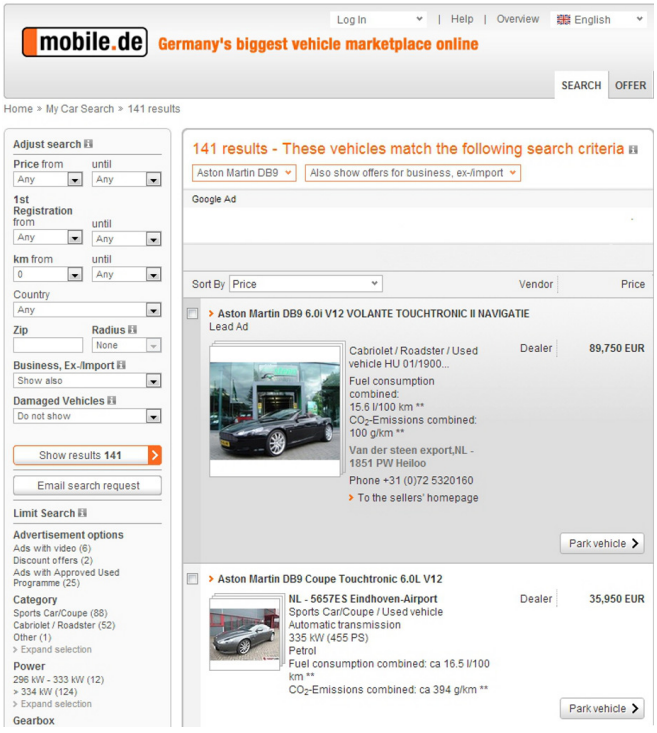

Fig. 7. Mobile.de Show Results Form

Furthermore, FilterGroup contains two UI Test Patterns: *Input* (Limit Search) and *MasterDetail* (Filter). These two UI Test Patterns are double connected with two SequenceWithMovedData (o[ ]≫) connectors meaning that there is data exchanging between them. The AdjustSearch is a *Find* UI Test Pattern and is responsible for testing the corresponding search.

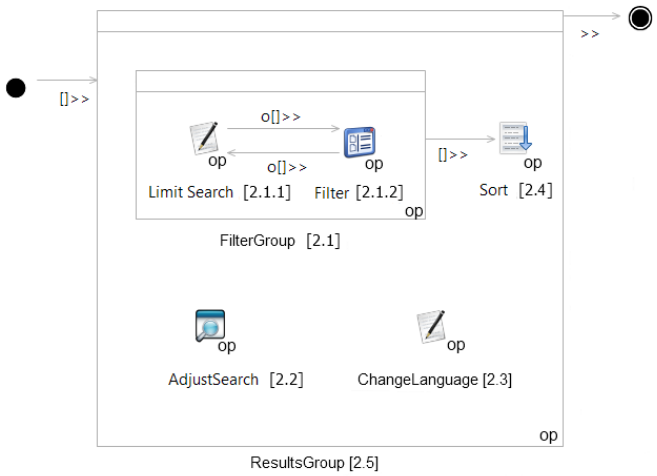

Fig. 8. Mobile.de Show Results Form Model

B. Generated Test Cases

After the model has been constructed, it is possible to generate test cases from it. These test cases are generated automatically by PARADIGM ME. Hence, for the given model of the *Mobile.de*, the test cases are then generated according to the following sequential steps: (1^{st}) test paths are created without detailing test strategies of the UI Test Patterns; (2^{nd}) test paths generated in the previous step are unfolded for encompassing the test strategies of each UI Test Pattern.

C. Test Paths

The test path at the higher level of the model is Init – 1 – 2 – End (Figure 4). The final test paths are constructed by expanding recursively each of those *Form* elements (this case 1 and 2). The Car Search *Form* element within Figure 4 is detailed in Figure 6. Such *Form* is comprised by an optional *Group* and a *Find* UI Test Pattern. Since the *Group* element (1.1) is optional, it is possible to navigate directly to the Find element (1.2). This means that there will be a test path containing only the Find element (1.2). According to Definition 3, the maximum number of test paths for the *Group* element (1.1) will be

$$\sum_{i=3}^{4} i! \cdot {}^{4}C_i + 1 = 49$$

The ShowResults *Form* within Figure 4 is detailed in Figure 8. It has *Init*, *ResultsGroup* and *End*. ResultsGroup (2.5) has four elements: AdjustSearch (2.2), ChangeLanguage (2.3), Sort (2.4) and FilterGroup (2.1). According to Definition 3, the number of test paths for the *Group* 2.1 is

$$\sum_{i=1}^{2} i! \cdot {}^{2}C_i + 1 = 5$$

The number of test paths for the ResultsGroup (according to Definition 3) without expanding FindGroup (2.1) is

$$\sum_{i=3}^{4} i! \cdot {}^{4}C_i + 1 = 49$$

From the 49 test paths of Group 2.1, 7 of them does not contain element 2.1. So, according to Definition 4, the number of test paths for the overall model in Figure 8 will be:

$$(49 - 42) + 42 \cdot 5$$

which is 217 test paths. These paths are the ones necessary to fulfill the goals identified earlier in this section, according to our methodology.

D. UI Test Patterns Configuration

After building the model, the Tester configures each test UI Test Pattern so the previously 217 test paths are expanded according to the test configurations defined for each UI Test Pattern which results in the final test cases that are needed to fulfill the test goals defined in Section V A.

PARADIGM ME allows manual insertion of test data. The configuration for the UI Test Patterns in the path "Init – 1.2 – 2.2 – End" and corresponding checks were:

- **UI Test Pattern 1.2** (Figure 6): {<FND_VF, {[Make, "Any"], [Model, ""]}, [Provide *Make*, Provide *Model*], #Results: 1.471.315, True>; <FND_NF, {[Make, "Bentley"], [Model, "Turbo RT"]}, [Provide *Make*, Provide *Model*], #Results: 0, True>};
- **UI Test Pattern 2.2** (Figure 8) : {<FND_NF, {[Price until, "Any"],[km until, "Any"]}, [Provide *Price until*, Provide *km until*], #Results: 0, True>; <FND_VF, {[Price until, "Any"], [Km until, "Any"]}, [Provide *Price until*, Provide *km until*], #Results: 1.471.315, True>};

During test execution, the correspondent checks for each configuration are performed. If one of those checks fails, the test case fails and an error is reported. For the above test path, the check "Results: 1.471.315" within test "FND_VF" of the UI Test Pattern 2.2 fails. After reviewing the problem reported, we realized that the number of elements of the search results set shown in page "Car Search" (Figure 5) is different from the number of the elements shown on page "Show results" (Figure 7) after submission when no search refinement is performed, as is the case of this test.

VI. Feasibility Study

The objective of this study (or experiment) is to assess the PBGT general approach, in order to ascertain whether it can be used by common people (not involved in the development of the language PARADIGM nor the development of its supportive environment) and to measure the amount of time required to begin getting benefits of the approach and start finding bugs in real applications. In particular, this case study was designed to answer the following research questions:

- **Time:** How long it takes to start using PARADIGM language and finding errors?

- **Expressive power:** Are the Base UI Test Patterns currently provided by PARADIGM enough for modeling real applications? Does the language have sufficient expressive power to model the testing goals that we propose?
- **Modeling environment usability:** Does the modeling environment needs improvement? In particular, does it allow building and configuring models easily?

A. Metrics

In order to answer the research questions above, we defined a set of metrics to be gathered during the execution of the experiment. In particular, we registered the time needed to introduce the language and the modeling environment to the students and asked the students to register the time taken for building and configuring the model. In addition, we asked the students to indicate all the problems they had to face during the experiment, for instance, doubts concerning how to model and configuring a specific testing goal and problems with the usability of the PARADIGM environment, such as, too many steps needed for configuration.

B. Subject Application

For this case study, we selected an Australian site – austra lian-charts.com/ [11], that in short, reflects the best selling music singles and albums in Australia.

This website shows the current top 50 and the best of all time singles and albums; allows searching for songs, albums, compilations and DVDs; shows a score based on reviews; allows new user registration and, afterwards, authentication; provides support for forums; among other functionalities.

In addition, this site has another functionality that allows to access sites for the same purpose (listing the best selling music) from other countries, such as, Finland, Germany, etc.

C. Procedure

We asked three students, who have never seen PARADIGM, to embrace the role Tester. The PARADIGM language and the modeling environment (ME) were introduced to the students during a 2-hour session. We clarified all students doubts and presented some modeling examples. Afterwards, we delivered the testing goals for them to model. Hence, the defined testing goals were:

1) test if the authentication is functioning correctly by displaying a message when the login fails;
2) test if it is possible to register new users and if messages are displayed for successful and unsuccessful registration;
3) test the dependency between the selection of a different country and the contents of a list with different options;
4) test the dependency between the selection of an element within a list of options and the content displayed in the middle panel (e.g., selecting "Top 50 Singles" displays "Of Monsters and Men – Little Talks" in the 6^{th} position; selecting "Search" displays a search area; selecting "Best of All Time" displays an area with the "best singles" since 1989);
5) test different searches (top, simple, extended) available (e.g., search "Song" with name "nothing" and check if the first artist is "Alesha Dixon"; search "Album" with the name "justice for all" and check if the first artist is "Metallica"; search "forum" without parameters and check if the result displays "show all topics").

The students then started the experiment, and subsequently, created the models to fulfill the testing goals and configured the UI Test Patterns (Behavior elements) present in the models. Next, they generated automatically the test cases via PARADIGM ME. Afterwards, they analyzed the test paths and documented the errors found.

D. Findings

All the students were able to build the model and fulfilled the expected testing goals, however they constructed different models. After a more detailed analysis on the models produced, we could conclude that the differences were related to the fact that they were using different levels of abstraction. One used more levels than the others, because he opt to use *Forms* detailed in other levels of detail. They spent, in average, approximately 44 minutes modeling and 31 minutes with configuration. All students were able to detect the same set of errors. Some of them were related to a search followed by an invalid login and other related with the dependency between UI elements, as follows:

1) Selecting "Top 50 Singles" and afterwards performing an invalid login shows a empty white page.
2) Performing a top search looking for *forum* and afterwards performing an invalid login, shows an error message: "Microsoft OLE DB Provider for SQL Server error...".
3) Performing a simple (or extended) search looking for songs with name "nothing" and performing an invalid login afterwards, does not show the invalid login message as expected.
4) In addition, changing the country to "United Kingdom" does not show "Search" neither "UK Charts" in the related area as what happens with other countries.

The students did register some complaints about the usability of the modeling environment. They complained that some configurations, in particular the configuration of the Find UI Test Pattern, demanded too many steps, which became boring. They also had difficulty in modeling a situation in which a *B* element appears after an optional *A* element, but that should only be run when *A* is executed. In fact, at present, for a path A-B the modeling environment generates test paths A; B; and A-B. The students wanted to generate only the paths A; and A-B not allowing the generation of a path in which B occurs alone.

We think that we can improve the language with an additional connector imposing a strict sequence between the connected elements. Right now, the connectors do not allow to invert the sequence between the elements, for instance, it does not generate a test path B-A when the elements are connected and B comes after A.

VII. Related Work

Our work borrows from prior work on UI and test patterns to build upon existing work on model-based GUI testing. In this section, we discuss prior research in these three areas.

A. UI Patterns

UI patterns, in our terms, represent generic graphical implementations of a given functionality. Although certain functionalities, such as "Login", can have different layouts, the aspect that is particular relevant for our work is the common behavior (that is repeated over time) described by these common implementations. The research of several UI Patterns was the entry point that led us in the direction of UI Test Patterns. The **Master/Detail** UI Pattern allows the user to stay in the same screen while navigating through items. In addition, it features two areas: master and detail. The content of the detail area is displayed based upon the current selection in the master area. One practical example of this pattern is iTunes [12]. The UI Pattern that provides a way for the user to enter data in the system, is known as the **Input Prompt** UI Pattern. Another popular UI Pattern is the **Auto-Complete**. It can be found in the Google search engine [13]. While the user types a word, a list of possible matches appear and then the user selects the desired item. The **Input Feedback** UI Pattern provides feedback to the user after submitting data. The goal is to notify users about errors that occurred during submission.

B. Testing Patterns

To the best of our knowledge, approaches have yet to appear concerning the matter of Pattern-Based GUI Testing. However, the concept of patterns has been used in the domain of software testing [14]. This approach consists in ensuring the implementation of a given design pattern (micro-architecture) corresponds to its design [15]. Thus, in this approach each pattern has associated a set of pattern test case templates (PTCTs). A PTCT represents a reusable test case structure designed to identify defects associated with application of the pattern. One drawback of this approach is that it requires deep knowledge over the structure of the actual system and full source-code access. This approach performs a static analysis over the code. Our approach (PBGT) performs a dynamic analysis over the GUI and because of that, it is independent from platform and programming language. This is the major difference between these approaches.

C. Model-Based GUI Testing

State machine models are the most popular models for testing, and have been effectively used to model GUIs [16], [17]. In the context of GUI models, windows and controls represent states, and user events represent transitions among states. However, when modeling a full scale GUI, the enormous set of states and transitions would be very high, turning the model very hard to work with due to its complexity. Additionally, state-charts' models can also be used for both modeling and generating test cases for GUIs [18]. The use of state-charts in GUI testing has not yet received great interest which is due to the lack of approaches and contributions to this matter. Furthermore, this approach has the same problem as the state machines. The size of the model will make difficult to work with as well as maintain it.

Labeled State Transition Systems (LSTSs) are an alternative approach to finite state machines. LSTSs are an extension of the Labeled Transition System (LTS) format by adding up labels to both states and transitions. A LTS consists of a set of states and a set of transitions among those states. These transitions are labeled by actions and one state is designated as the initial state. This approach uses keyword and action word techniques. In one hand, action words describe user events with a high level of abstraction. On the other hand, keywords correspond to key presses and menu navigation. The main idea is to provide simple means for the domain experts so that they can design test cases with action words even before the system implementation. This simplifies the work of testers since programming skills will not be required. According to [19], by varying the order of events it becomes possible to find previously undetected events. In this approach, formal models can be built by using tools from a custom (Tampere Verification Tool - TVT) verification toolset [19]. This method is very formal in a way that models must be created by hand, and its syntax is not straightforward to use. In addition, the toolset lacks support and comes into sight that LTSs are less expressive than finite state machines.

Other approach makes use of Event Flow Graphs (EFG) to create a GUI model [20]. The idea behind this concept is to capture the flow of events, featuring all possible event interactions in the UI. The EFG is comprised by nodes and edges. An edge from one node to another means that the second node can be executed immediately after the first one. EFG edges are not labeled, nor do EFGs have states. Furthermore, models are generated directly from an executable by means of a GUI Ripping Tool. As best of our knowledge, no tool exists to model an EFG by hand so it is not possible to edit the ripped models. For complex applications GUI ripping falls short in generating the full model [21]. Current ripping tools are architecture dependent; versions for web, desktop, mobile, .NET platforms are available.

Another approach to create GUI models is to use Event Sequence Graphs (ESG) [22], [23], which are extensions of EFGs in that ESGs allow multiple start and final nodes. In order to model such behavior, pseudo entry nodes denoted by "[", and pseudo finish nodes denoted by "]", are used to mark start and finish events, respectively.

Other approach features a GUI mapping tool [24], where the GUI model is written in Spec# with state variables to model the state of the GUI and methods to model the user actions on the GUI. However, the effort required for the construction of Spec# GUI models is too high. In the context of this work [24], another attempt was made to reduce the effort to construct a GUI model. A visual notation entitled VAN4GUIM [25] was designed and translated to Spec# automatically. The aim was to have a visual front-end that could hide Spec# formalism details from the testers.

Additionally, task models can be used in MBGT [26]. The use of task models reduces the effort in creating test oracles. The approach described in [26] focuses on user errors, and examines the feasibility of using task models to test GUIs against erroneous user behavior. Moreover, it uses ConcurTaskTrees (CTT) as the task modeling notation and offers tool support (CTT Model Transformation Tool - CMTTool). The aim of this tool is to generate several mutants from the task model for testing purposes.

VIII. Conclusions

This paper presented a Pattern-Based approach for GUI modeling and testing (PBGT). This testing approach is based on a DSL language, called PARADIGM, which aims to simplify the modeling process by promoting reuse. Test cases are automatically generated from PARADIGM models.

In the future, we intend to experiment our approach in real industrial environments projects, within different contexts, like developments from scratch and maintenance. We also intend to study the possibility to extend the set of Base UI Test Patterns available within PARADIGM language. In addition, the number of connectors may also be extended, for example, to address the need identified by the students, helping to increase the expressiveness of the PARADIGM language.

Acknowledgments

This work is supported by (1) European Regional Development Fund (ERDF) through the COMPETE Programme (operational programme for competitiveness), (2) National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554, and (3) US National Science Foundation (NSF) via grant number CNS-1205501.

References

[1] T. Neil, "12 Standard Screen Patterns," http://designingwebinterfaces.com/designing-web-interfaces-12-screen-patterns, accessed November, 2012.

[2] Patternry, "UI Design Patterns and Library Builder," http://patternry.com, accessed November, 2012.

[3] J. Tidwell, *Designing Interfaces*. Sebastopol, CA: O'Reilly, 2011.

[4] A. Toxboe, "UI Patterns - User Interface Design Pattern Library," http://ui-patterns.com, accessed November, 2012.

[5] M. van Welie, "Interaction Design Pattern Library," http://www.welie.com/patterns, accessed November, 2012.

[6] Yahoo!, "Yahoo! Design Pattern Library," http://developer.yahoo.com/ypatterns, accessed November, 2012.

[7] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Softw. Pract. Exper.*, vol. 39, no. 15, pp. 1253–1292, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1002/spe.v39:15

[8] F. Paternò, C. Mancini, and S. Meniconi, "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," in *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction*, ser. INTERACT '97. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 362–369. [Online]. Available: http://dl.acm.org/citation.cfm?id=647403.723688

[9] N. Skrypuch, "Eclipse Modeling – EMF – Home," http://www.eclipse.org/modeling/emf, accessed November, 2012.

[10] "mobile.de - Germany's Biggest Vehicle Marketplace Online. Search, Buy and Sell Used and New Vehicles," http://www.mobile.de, accessed December, 2012.

[11] S. Hung, "Australian charts portal," http://australian-charts.com, accessed February, 2012.

[12] A. Inc., "Apple – iTunes – Everything you need to be entertained." http://www.apple.com/itunes, accessed February, 2012.

[13] "Google," http://www.google.com, accessed February, 2012.

[14] A. Pehmöller, F. Salger, and S. Wagner, "Patterns for testing in global software development," in *Proceedings of the 13th International Conference on Quality Engineering in Software Technology*, 2010.

[15] N. Soundarajan, J. Hallstrom, G. Shu, and A. Delibas, "Patterns: from system design to software testing," *Innovations in Systems and Software Engineering*, vol. 4, pp. 71–85, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11334-007-0042-z

[16] F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces," in *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ser. ISSRE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 34–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851028.856279

[17] L. White and H. Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences," in *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ser. ISSRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 110–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851024.856239

[18] H. Reza, K. Ogaard, and A. Malge, "A model Based Testing Technique to Test Web Applications Using Statecharts," in *Proceedings of the Fifth International Conference on Information Technology: New Generations*, ser. ITNG '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 183–188. [Online]. Available: http://dx.doi.org/10.1109/ITNG.2008.145

[19] A. Kervinen, M. Maunumaa, T. Päakkönen, and M. Katara, "Model-based testing through a GUI," in *In Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), number 3997 in Lecture Notes in Computer Science*. Springer, 2006, pp. 16–31.

[20] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage Criteria for GUI Testing," in *In Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*. ACM Press, 2001, pp. 256–267.

[21] P. Aho, N. Menz, T. Räty, and I. Schieferdecker, "Automated Java GUI Modeling for Model-Based Testing Purposes," in *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations*, ser. ITNG '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 268–273. [Online]. Available: http://dx.doi.org/10.1109/ITNG.2011.54

[22] F. Belli, M. Beyazit, and N. Güler, "Event-Based GUI Testing and Reliability Assessment Techniques – An Experimental Insight and Preliminary Results," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 212–221. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.59

[23] A. Datentechnik, "TSD – Test Suite Designer," http://designingwebinterfaces.com/designing-web-interfaces-12-screen-patterns, accessed November, 2012.

[24] A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing." in *ICFEM*, ser. Lecture Notes in Computer Science, K.-K. Lau and R. Banach, Eds., vol. 3785. Springer, 2005, pp. 450–464. [Online]. Available: http://dblp.uni-trier.de/db/conf/icfem/icfem2005.html#PaivaFTV05

[25] R. M. L. M. Moreira and A. C. R. Paiva, "Visual Abstract Notation for GUI Modelling and Testing – VAN4GUIM." in *ICSOFT (SE/MUSE/GSDCA)*, J. Cordeiro, B. Shishkov, A. Ranchordas, and M. Helfert, Eds. INSTICC Press, 2008, pp. 104–111. [Online]. Available: http://dblp.uni-trier.de/db/conf/icsoft/icsoft2008-2.html#MoreiraP08

[26] A. Barbosa, A. C. Paiva, and J. C. Campos, "Test case generation from mutated task models," in *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS '11. New York, NY, USA: ACM, 2011, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/1996461.1996516