Received: 17 May 2014 | Revised: 10 November 2016 | Accepted: 2 January 2017
DOI: 10.1002/stvr.1629

WILEY

RESEARCH ARTICLE

Pattern-based GUI testing: Bridging the gap between design and quality assurance

Rodrigo M. L. M. Moreira[1] | Ana Cristina Paiva[1] | Miguel Nabuco[1] | Atif Memon[2]

[1]INESC TEC and Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal
[2]Department of Computer Science, University of Maryland, College Park, MD, USA

Correspondence
Rodrigo M. L. M. Moreira, INESC TEC and Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal.
Email: pro08007@fe.up.pt

Funding Information
European Regional Development Fund (ERDF) National Funds, Grant/Award Number: FCOMP-01-0124-FEDER-020554; US National Science Foundation (NSF), Grant/Award Number: CNS-1205501

Summary

Software systems with a graphical user interface (GUI) front end are typically designed using user interface (UI) Patterns, which describe generic solutions (with multiple possible implementations) for recurrent GUI design problems. However, existing testing techniques do not take advantage of this fact to test GUIs more efficiently. In this paper, we present a new pattern-based GUI testing (PBGT) approach that formalizes the notion of UI Test Patterns, which are generic test strategies to test UI patterns over their different implementations. The PBGT approach is evaluated via 2 case studies. The first study involves 2 fielded Web application subjects; findings show that PBGT is both practical and useful, as testing teams were able to find real bugs in a reasonable time interval. The second study allows deeper analysis by studying software subjects seeded with artificial faults; the findings show that PBGT is more effective than a manual model-based test case generation approach.

KEYWORDS

GUI testing, model-based GUI testing, pattern-based GUI testing, UI test patterns

1 | INTRODUCTION

In recent decades, graphical user interfaces (GUIs) have been gaining importance and popularity. They assist us in our daily lives to facilitate tasks and can be seen in several different contexts (smart phones, tablets, personal computers, televisions, tablets, GPSs, and so on). Graphical user interfaces are the means through which end users perceive the systems.

They are able to perform tasks through GUIs (via events) in diverse ways. Although this flexibility has improved usability, it constitutes a challenge for software testing because testers have to decide to either check all sequences of events or check only a subset. With automated software testing, it is possible to reduce the effort required to test GUIs. Nevertheless, despite the progress made in automated testing tools over the last decade, manual testing still represents most testing efforts in practice.

There are several popular testing techniques, such as random testing [1] and capture/replay (CR) [2, 3] that can be applied to test GUIs. The former type of tools are only able to find certain type of errors, while the latter type is not able to automate test case (TC) generation. Capture/replay are more useful to perform regression testing. Other testing techniques, called model-based testing (MBT) techniques [4–7] are able to generate TCs from models. However, most of the time, MBT are not developed with reuse concerns as a way to diminish testing effort when similar behavior is present in different GUIs or even within the same GUI under test. Testers start from scratch and do not have a mechanism to support reuse.

In this paper, we present a new testing approach that leverages user interface (UI) patterns to promote reuse in MBT. Consider a very common UI pattern developed around the common authentication mechanism in software systems that aim to provide access to certain restricted functionalities, upon successful authentication. Independent of the specific implementation, a typical authentication mechanism features at least 2 input fields (username and password) and 1 submit button. The actions performed by the user are provide login, provide password, and press submit. This is a recurrent behavior, independent of the implementation. Further, the user is able to provide valid and/or invalid inputs, such as valid username and password; invalid username and password (due to valid username and invalid password; due to invalid username and valid password; due to username left blank; or due to password left blank). Yet, depending on the implementation, the outcomes may differ. In some systems, if the authentication fails, an error message may be displayed or a pop-up window is displayed containing the error message. Even the error messages can vary, according to the provided input: "Invalid email," "Re-enter your password," "Invalid login," among others. Successful authentication also stipulates

other particularities: the user may be redirected to a different location or stay in the same location but with a different view (UI) enabling new higher privileged features for the user. Such UI patterns have traditionally been used for GUI design. As GUIs grow in size and complexity, so does the usage of UI Patterns.

Until recently, testing GUIs for their functional correctness has ignored UI Patterns. In the context of our work, in our terms, we refer to UI Patterns as solutions of given functionalities, to solve common GUI design problems. User interface patterns can be implemented in a multitude of ways, yet maintaining common behavior. By observing the latter, in the context of testing, we realized that GUIs that have similar design, based on a given UI Pattern, should share a common testing strategy. Pattern-based GUI testing (PBGT) is a new emerging model-based GUI testing (MBGT) approach that aims at systematizing and automating the GUI testing process, by benefiting from **UI Test Patterns** and therefore promoting reuse of GUI testing strategies. A UI Test Pattern provides a generic way to test the different implementations of UI Patterns and defines a reusable and configurable test strategy, to test a GUI that was implemented using a set of UI Patterns. User interface test patterns are elements within a domain-specific language (DSL) (PARADIGM) from which it is possible to build models describing the GUI testing goals in a higher level of abstraction. Pattern-based GUI testing has the ability to automatically generate TCs from those models and execute them over a GUI. As it is common practice in GUI testing, PBGT considers the entire software as a black box, exercising its input space via the GUI elements and observing its correctness via the state of its widgets.

Besides providing a set of UI Test Patterns, PBGT allows defining new ones by combining the existing ones and promoting even further reuse. When changes occurred in the system, the testing goals described by the model can be updated and then the whole forward testing process is automatic.

The PBGT approach was empirically evaluated by conducting 2 case studies. In the first study, the goal was to measure the overall effectiveness of the PBGT approach and the effort required to start using it. This study was conducted on 2 fielded systems. Results indicate that testers were able to identify failures on both systems. Within an average of 32.75 minutes, the testers were able to use PBGT and start finding failures. The purpose of the second case study is to compare PBGT (MBT) approach with random testing and manual model-based generation of TCs (both departing from the same specification of the testing goals) regarding their ability to find failures and testing time spent on activities prior to executing the tests. Results indicated that, even though PBGT took more time to configure test models, it could find more failures than manual model-based generation tests and more than random testing. In this study, we also compare/evaluate PBGT's different test strategies (Default, Random, and Invalids). Results indicated that the Random strategy could find more failures, but it also took more time.

In summary, PBGT approach makes the following contributions:

1. **Reusability and reduced costs.** By promoting reusability with PBGT, we are able reduce the modeling and testing efforts allowing to save cost in GUI testing, either by reusing UITPs, combinations of UITPs or entire models to test other websites.
2. **Reduced efforts.** Models can be crafted and configured in short time, when compared with other MBGT approaches [8].
3. **Goal oriented.** Modeling within PBGT is directly focused on modeling the testing goals instead of modeling the behavior of the application, which is mainly the approach followed by other MBGT tools.
4. **Low maintenance and evolutionary language.** Pattern-based GUI testing provides the capability to extend the initial set of UI Test Patterns, to adjust current test strategies (or create new ones) to fulfill new UI trends.
5. **Freedom of choice.** The input data for testing is provided by the tester. Pattern-based GUI testing does not force the tester to use a particular technique, so instead, the tester has the freedom to use any technique, such as equivalence class partition or boundary value analysis.
6. **Tool support.** Pattern-based GUI testing is supported by a tool that provides capabilities for modeling GUI's testing goals using a DSL (PARADIGM), and providing functionalities such as UI Test Pattern configuration, TCs generation, TC execution, and test coverage analysis.

Our previous work presented our vision of UI test patterns, examples such as (Login, Master/Detail, and Sort), and preliminary studies validating the PBGT approach and its capability in finding failures [6, 9, 10]. We now build upon our previous work by providing additional material. First, we provide a description of UI Test Patterns according to best practices and guidelines from the field of Pattern Languages. We present patterns in great detail (Input, Find, and Call) and a new pattern (the option set pattern). Second, we provide more details of our experimental evaluation detailing the steps performed in the experiment, presenting the testing goals used, the test models developed for the Australian Chart application, the configuration data, examples of the TCs generated, and threats to validity. Third, we do a detailed comparison between PBGT testing methodology and random testing and manual generation of TCs; some results of this experiment were published in [10], but we now present additional details: we detail the procedure, list the testing goals, present the configuration data used, and present a comparison of testing techniques for 2 other subject systems (TaskFreak and iAddressBook). Finally, we present a detailed discussion of the results achieved by the 2 experiments detailed in the paper.

The remainder of this paper is structured as follows. Section 2 presents the state of the art review on the topic of GUI testing. Section 3 provides a synopsis on the PBGT approach and its scope. Section 4 presents UI Test Patterns formally, their usage, configuration concerns, and TC generation methods for PBGT. Section 5 demonstrates the empirical evaluation performed. Section 6 discusses results from the case studies. Finally, Section 7 draws conclusions.

2 | STATE OF THE ART

Because of the broad usage of GUIs in numerous software systems, GUI testing has become an area of increasing importance to software development. Graphical user interface testing is often perceived as a challenging task. It is difficult, extremely time consuming, and

costly, with scarce tools and techniques available, to assist the testing process.

Current GUI testing methods still require considerable manual efforts. Manual testing is a laborious activity, specifically error prone, and tedious. In addition, testers are required to be patient, perceptive, creative, and skillful. As GUIs grow and, consequently, become more complex, they make even further manual GUI testing a daunting task to comply. However, endeavors have been made to automate the GUI testing process. Automation ensures that tests are performed regularly and consistently, resulting in early error detection that leads to enhanced quality. With a proper automated GUI testing process, more TCs can be conducted and more faults can be found within less time.

2.1 | Capture/replay

Capture/replay is a popular GUI testing approach that attempts to automate test execution. This approach is supported by a set of tools* that provide the capability to record user actions, such as mouse motions, mouse clicks, and keyboard inputs, in test scripts, while interacting with the GUI. Based on the test scripts, the user actions can later be replayed, when required, with the goal to verify if the reaction of the system matches the expected result [11]. The test scripts are specified in a particular scripting language provided by the CR tools.

One of the drawbacks of this approach is the cost of script maintenance. If the GUI changes, it will cause a large number of previously recorded tests to break. This means that tests will need to be manually edited and later rerun. Such difficulty to adapt to small changes indicates a maintenance problem that often leads CR method to be discarded after several software releases.

Yet there are advantages to using CR tools. Testers can use them as fast means to capture the first test and therefore get early feedback. Furthermore, testers can use this method on early prototypes at design time. Since the GUI's functionality is not yet implemented, there are no bugs that interrupt the capturing of test scripts [12]. Moreover, the recorded test scripts are suitable for regression testing.

2.2 | Random testing

Random testing is also recognized as *stochastic* testing or *monkey* testing. The term monkey is referred to any form of automated testing performed randomly without any typical user bias [1].

The idea is to have *someone* (ie, a monkey) who does not know how to use the application, interacting with the latter, to perform mouse clicks and keyboard strokes, arbitrarily. Microsoft reported that 10% to 20% of bugs are found via test monkeys.

This method distinguishes 3 types of monkeys. "Dumb" monkeys do not have any idea about the system, nor its state. In addition, they are not aware of which inputs are legal or illegal. The downside is that these monkeys are unable to recognize a failure when they face one. Their key goal is to try to crash the system under test (SUT). Other monkeys, referred as "semismart" monkeys, are able to recognize a bug when they encounter one. "Smart" monkeys have certain awareness about the application they are testing. They obtain their knowledge from a state table or model of the SUT. They traverse the state model and choose the legal steps that allow to move forward in each state. Nevertheless, smart monkeys are most costly to develop.

2.3 | Model-based testing

Model-based testing has been gaining considerable interest in software testing [7, 13]. Model-based testing has been used successfully to testing GUIs. A lot of research has been conducted since the past decade. Model-based testing is a software testing technique upon which TCs are derived from a model that describes functional aspects of the SUT [14]. It allows checking the conformity between the implementation and the model of the SUT, introducing more systematization and automation into the testing process. The test generation phase is based on algorithms that traverse the model and produce tests as desired.

One popular approach uses event-flow graphs (EFGs) to create a GUI model [15]. The idea is to capture the flow of events, featuring all possible event interactions in the UI. The EFG is comprised of nodes and edges. An edge from 1 node to another means that the second node can be executed immediately after the first one. The 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 (GUITAR [4]).

Other approach uses labeled state transition systems, action words, and keywords, with the goal to describe a test model as an Labeled Transition System (LTS), where its transitions correspond to action words [16]. An LTS consists of a set of states and a set of transitions among those states. Action words describe user events with a high level of abstraction, and keywords correspond to key presses and menu navigation. The idea is to provide simple means for the domain experts so that they can design TCs with action words even before the system implementation. This simplifies the work of testers since programming skills will not be required. According to the authors, by varying the order of events, it becomes possible to find previously undetected events.

Finite state machines ([17]) are one of the most popular used models for software design and software testing. Miao et al. [18] proposed an finite state machine–based approach (called GUI test automation model), with the goal to automate all the tests of the EFG-based approach, aiming to provide more automation concerns and thus solving limitations of the EFG approach. They indicate that EFGs are not able to model certain scenarios where GUI objects are modified dynamically: (1) changing the visibility of a GUI object, which depends on another objects' state (checkbox), via code will lead to an event being undefined; and (2) toggling of the visibility property value of a panel can cause several events to be uncovered or concealed. These events cannot be modeled and therefore cannot be tested. Additionally, state-charts models can also be used for both modeling and generating TCs for GUIs [19]. The use of state-charts in GUI testing has not yet received great interest that is due to the lack of approaches and contributions to this matter.

Petri nets can also be used in GUI testing. Reza et al. [5] propose a new model-based approach to test the structural representation of GUIs using hierarchical predicate transitions nets (HPrTNs). In their terms, HPrTNs are high class of Petri nets. They justify their usage due to the capability of HPrTNs to recognize and treat events (desirable

*http://www.ranorex.com/; http://www.seleniumhq.org/; https://www.ibm.com/developerworks/rational/products/robot/

and undesirable) and also states (desirable and undesirable behaviors). With HPrTNs, it is possible to model the planned behavior of GUIs and generate TCs from the model. Furthermore, authors also propose new coverage criteria for GUIs using HPrTNs.

Graphical user interfaces can be modeled with Spec# [20] describing actions performed by the end user and the expected outcome of each user action. Hence, GUI Spec# models are the input to Spec Explorer [21] with an MBT tool (GUI Mapping Tool [22]) that is able to generate and execute TCs automatically. The efforts in writing these models in Spec# is high. Therefore, to diminish this effort and to hide the formalism details from testers, Moreira et al. [23] developed a graphical visual notation entitled VAN4GUIM (Visual Abstract Notation for GUI Modeling and Testing), providing tool support. In addition, in this approach, models are translated automatically to Spec#, according to a set of rules. However, the model does not feature all GUIs behavior. Additional behavior needs to be included manually in the generated Spec#. After the Spec# model is completed, it will be used as input to Spec Explorer with GUI Mapping Tool.

2.4 | Discussion

Automated GUI testing involves performing a set of tasks automatically and then comparing their result with the expected output and, the ability to repeat the same set of tasks multiple times with different input data, maintaining the same level of accuracy. Furthermore, it has become tremendously important as GUIs become progressively more complex and popular. Test automation allows performing a set of testing tasks, without user intervention, with possible different test input data, which may increase the code coverage without significant additional effort. Current software testing tools and techniques suffer from lack of generic applicability and scalability. Despite the term *automated testing*, most of the approaches still require considerable manual efforts.

Capture/replay, although popular, faces some drawbacks: test data is hard-coded in the scripts; by default, it is not possible to compare expected output with the obtained output; scripts are hard to manage; the recapture takes the same time as the original capture; and the scripts are required to be modified upon small changes in the UI.

Regarding MBT tools, they also require significant modeling effort and have scalability problems. Finite state machines suffer from the enormous number of states and transitions. In addition, an increase on the complexity of the system would cause an explosion in the number of states and transitions. State-charts also suffer from the state explosion problem. Finite state machines cannot represent concurrency, since only 1 state can be active at a time.

Event-flow graphs are one of the most successful model-based techniques. Yet the generated graphs are too large and thus have a large testing suite. Further, it is complicated to test specific UI parts and to handle prioritization [24]. For rather complex applications, GUI ripping will have difficulties in generating the full model.

Petri nets provide a different perspective on the field of GUI modeling and testing. However, authors [5] have not proven the feasibility of their approach, nor provided a case study to support their view. Petri nets can be pointed as a good resource to model GUIs with the goal to represent concurrency. However, classic Petri nets do not support hierarchy [19].

In summary, the existing MBT approaches still need to increase the level of abstraction of their models and promote reuse to significantly reduce the modeling effort and the overall effort in software testing.

It is known that there are several pattern languages to describe the reusable elements of user-computer interaction [25]. Most of today, GUIs are built around these patterns, and thus, their behavior is expected to vary only slightly. So, if we take into account that most GUI behavior is recurrent, we are able to devise recurrent schemes to test that behavior. These elements would constitute a catalog of patterns for GUI testing, which although not aiming for completeness; it specifically targets common, recurrent behavior of software. Our first publication following this line of reasoning was [26]. The work presented in this paper makes additional contributions for this work.

3 | OVERVIEW

Graphical applications are developed using sets of windows and widgets. A widget represents a graphical element that is responsible for describing a specific behavior and functionality. These widgets can be implemented in different ways, and despite the fact that their layout may vary in implementation, their behavior across different implementations is typically the same. We name these widgets *UI Patterns*. User interface patterns, by our definition, represent generic graphical implementations of a given functionality. Since certain functionalities 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 [6].

One UI Pattern that is often seen in several software applications is the Master/Detail UI Pattern. This UI Pattern allows the user to navigate through items, while visualizing the whole UI. It consists of 2 areas: master and detail. The content of the detail area changes according to the selection of a specific item from the master area. iTunes and Finder (Figure 1), from the Mac OS, are 2 examples that feature the Master/Detail UI Pattern (a tutorial on how to implement this UI Pattern can be found in Mac Developer Library [27]).

The iTunes application features multiple optional selections for songs, albums, artists, genres, playlists, among others. These items represent the master area. Upon selection, the area below, ie, the detail, changes in respects to the item from the master. For instance, if the user selects "Albums" the detail will display the albums covers, titles, and artists, in mosaic. However, if the user selects "Artists" the layout and structure for the detail will be modified. The same behavior happens with Finder. The user is able to navigate through the UI, staying on the same screen, accessing the folders contents. This application features a number of instantiations of the Master/Detail UI Pattern arranged in a hierarchy.

In the same way that developers and designers make use and are assisted by UI Patterns during GUIs development, it makes sense to give support to testers during their activity too, which, in this case, can be done by providing generic test strategies to test those UI Patterns. We developed a new approach called PBGT, which goal is to provide

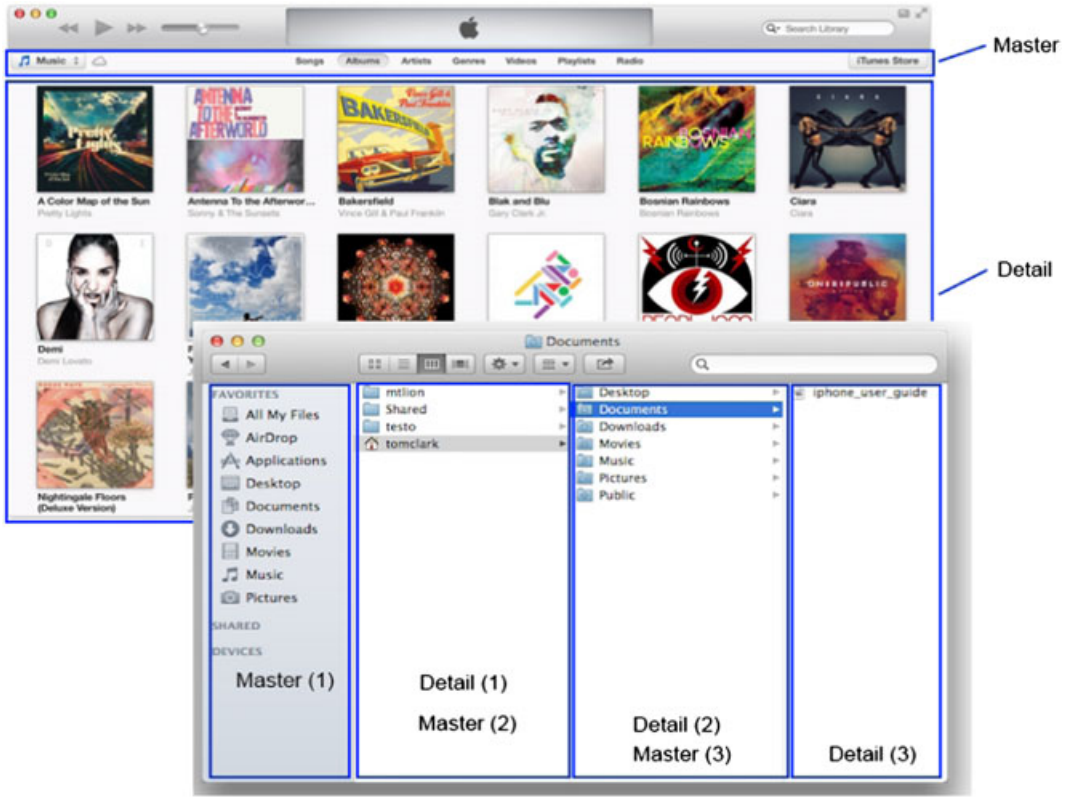

FIGURE 1 iTunes (top) and Finder (bottom) examples (sources from *www.apple.com*)

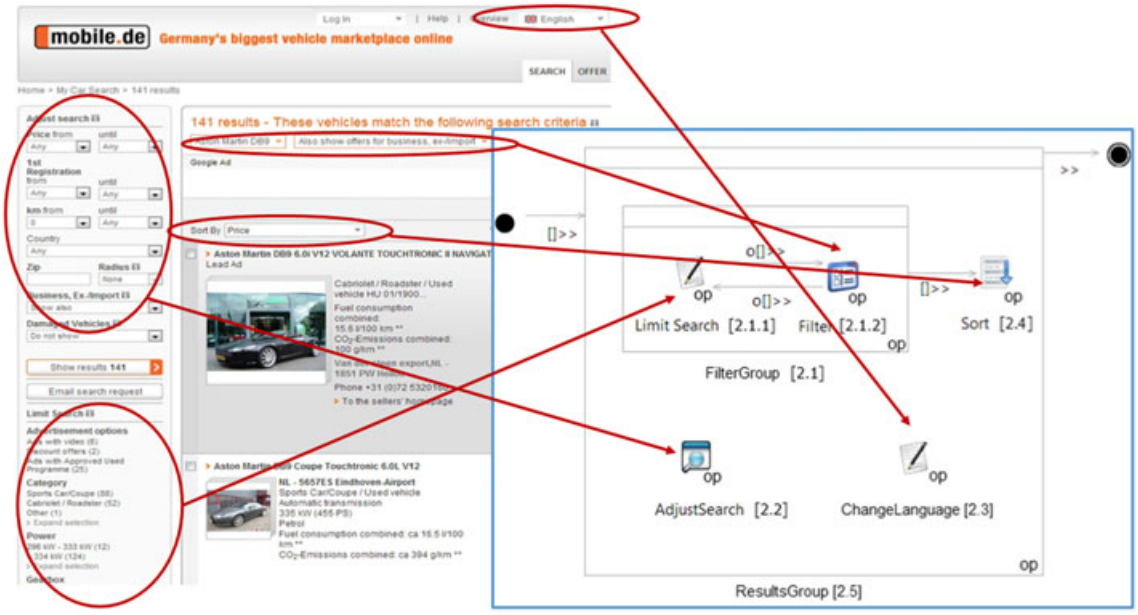

FIGURE 2 Selling cars

generic test strategies with different configurations to allow testing different implementations of a given UI Pattern and therefore promoting reuse. Consequently, we came up with the notion of *UI Test Patterns*. A UI Test Pattern provides a way to test UI Patterns across its several implementations and particular demands.

As an example, a UI Test Pattern defining a test strategy for the Master/Detail UI Pattern would have a testing goal (change master), which tests if changing the master, the detail updates accordingly. The sequence of actions to perform would be "select master." Regarding the checks to perform could be "detail has value X," "detail does not have value X," and "detail is empty." For this testing goal, the tester needs to define the input data for master and for detail. Then, the tester selects the checks to perform during TC execution and defines the corresponding precondition. A tester can use multiple times the same testing goal providing different configurations, ie, different input data and checks to perform.

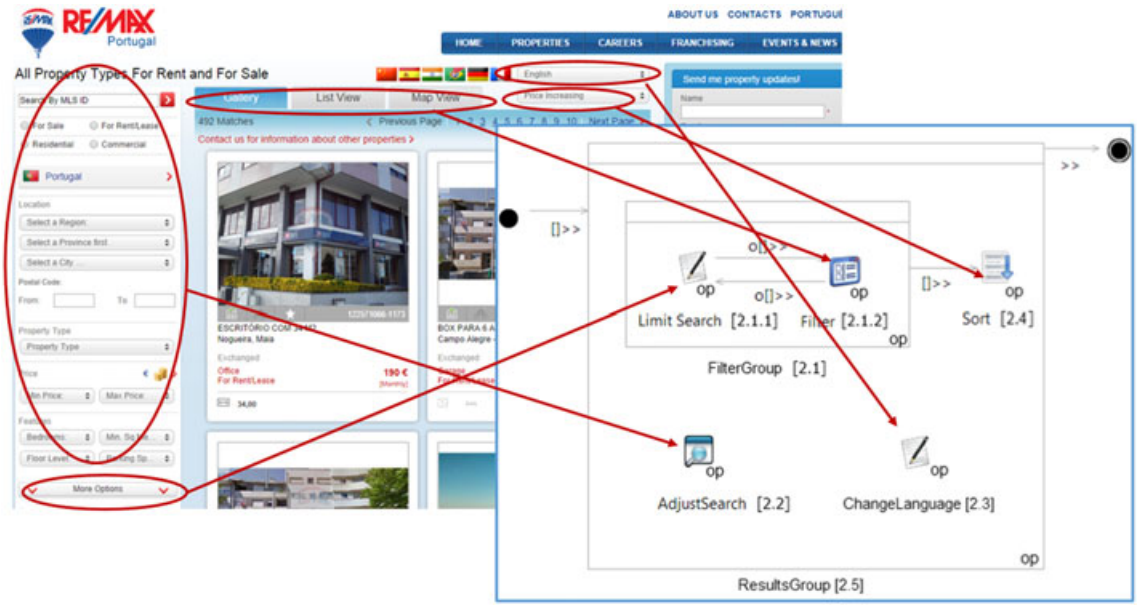

FIGURE 3 Selling houses

Graphical user interfaces feature UI Patterns that can be implemented in a multitude of ways. User interface test patterns are able to test UI Patterns across their several different implementations. Within PBGT, UI Test Patterns are classified as Base UI Test Patterns or High-Order UI Test Patterns. The Base UI Test Patterns represent the "starter kit" of UI Test Patterns that testers are able to use to test GUIs. The High-Order UI Test Patterns are seen as extensions of the Base UI Test Patterns. The higher-order UI Test Patterns are composed by 2 or more patterns. They allow the definition of a new set of UI Test Patterns that can be reused on different models. These higher-order patterns can be built by extending PARADIGM language increasing the initial set of patterns provided or by defining models inside Forms (structural element) that can be reused in other models to test other websites. For that the tester has to map the model elements to the new GUI elements within the other website to test. Figures 2 and 3 illustrate this situation where the same model (Form) is used to test 2 different websites: one to sell cars and another to sell houses.

3.1 | PARADIGM language

Pattern-based GUI testing requires a model to be used specifically for this domain. PARADIGM is a DSL developed to allow building models for being used within PBGT. Our previous work [8] describes the activities followed to build the PARADIGM DSL having as basis a set of guidelines and best practices. In addition [28] describes a new iterative approach using Alloy [29] to tune language constraints.

We also conducted a set of case studies to compare the modeling efforts of PARADIGM with other modeling approaches, such as Spec# [20] and VAN4GUIM [23]. The results showed that in an universe of 6 subject systems, PARADIGM took on average, 33% less time (in minutes) than VAN4GUIM and about 24% less than Spec#, in crafting and configuring models. The main reason for these results is the expressive power of PARADIGM, which allows describing the testing goals at a higher level of abstraction requiring less modeling effort.

The PARADIGM language metamodel is depicted in Figure 4. A model written in PARADIGM, for the PBGT domain, is a graph with nodes (*Elements*) and arrows (*Connectors*). It has 2 mandatory nodes: an *Init* and an *End* node (where the model begin and ends, respectively). Both *Init* and *End* are specializations of the abstract entity *Element*. Between these 2 nodes, there are several types of other elements present: UI Test Patterns, *Groups*, and *Forms*. Each UI Test Pattern can be either mandatory or optional, and it contains a specific ID, a number that identifies the element in the model. Models can be structured in different levels of abstraction. For instance, as models grow it is possible to use structuring techniques, to handle different hierarchical levels, so that a model A can "live" inside a model B. *Structural Form* elements were created specifically for this purpose. A *Form* (*Structural* element) embodies a model (or submodel), with an *Init* and *End* element. *Groups* are also structural elements but they are used to gather elements that may be executed in any order. *Behavioral* elements represent the UI Test Patterns that define strategies for testing the UI Patterns.

3.2 | The PBGT architecture

Pattern-based GUI testing supporting tool [30] is built on top of Eclipse Modeling Framework [31] and has a set of components described as follows:

- **PARADIGM** – A DSL for building GUI test models based on UI Test Patterns;
- **PARADIGM-ME** – The PARADIGM Modeling Environment (Figure 5) for building and configuring models written in PARADIGM language. On the right side is the palette with all the elements and connectors of the language. On the left is the modeling area and below the properties tab;
- **PARADIGM-TG** – A TC generation tool that builds TCs from PARADIGM models;
- **PARADIGM-TE** – A TC execution tool to execute automatically the generated test cases over the Web application under test, analyze the coverage, and generate reports with the results of the TC execution.

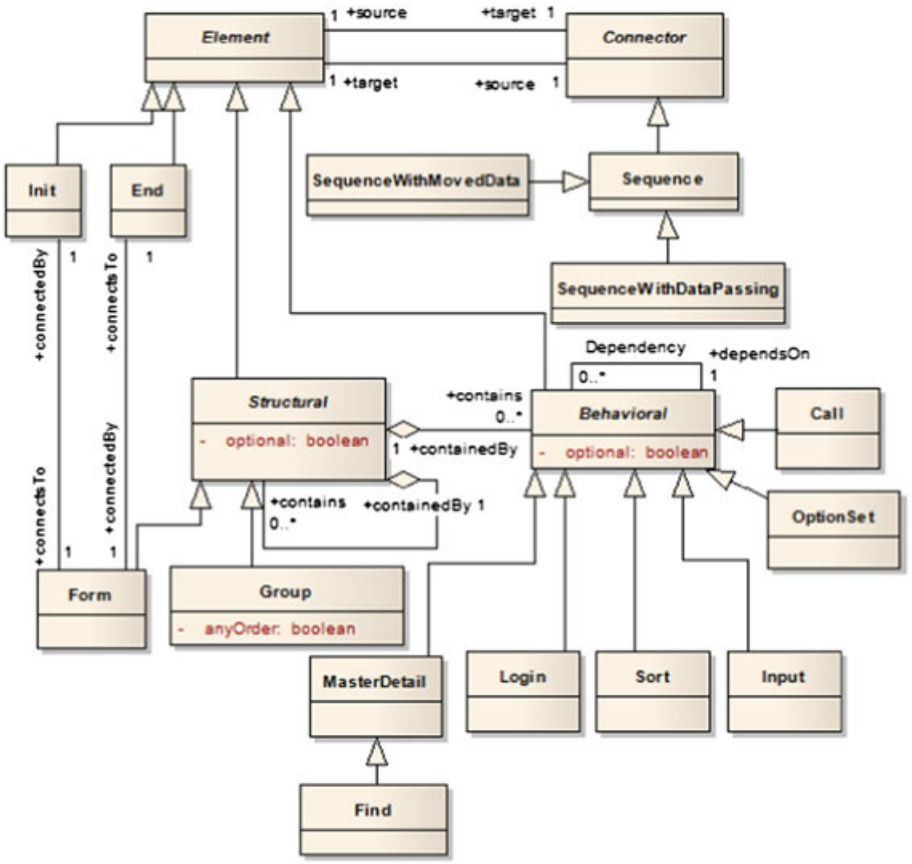

FIGURE 4 PARADIGM Language metamodel

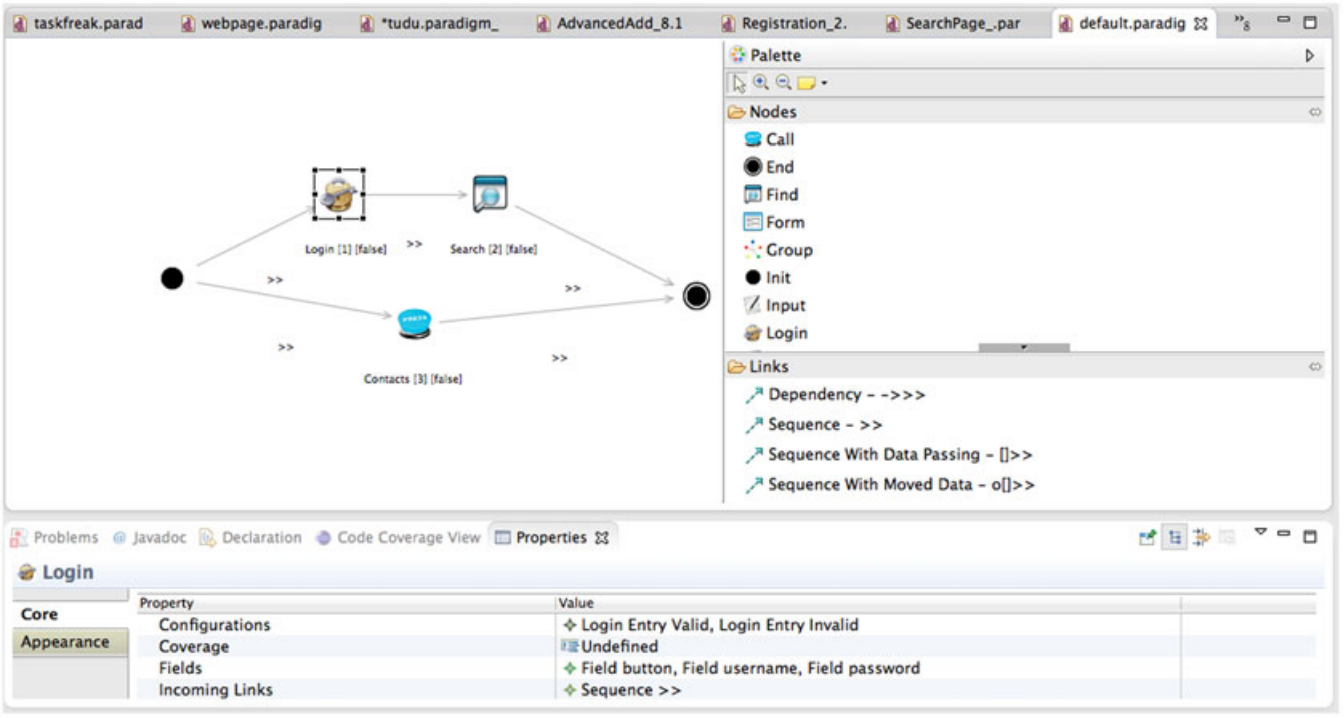

FIGURE 5 PARADIGM Modeling Environment

- **PARADIGM-RE** – A reverse engineering component whose purpose is to extract PARADIGM models from Web pages, without requiring access to the source code.

To automatically execute TCs over a Web application, PARADIGM-TE uses Selenium [32] and Sikuli [33]. When Selenium cannot identify the GUI control in which to act during TC execution, the tool uses Sikuli,

which is a visual technology to automate and test GUIs using screenshot images.

3.3 | The PGBT methodology

The PBGT methodology emerges as a new approach towards MBGT and aims to promote reuse of GUI testing strategies. This approach allows to combine UI Test Patterns and to build new ones. Someone who aims to adopt PBGT in a company may use it "as is" or extend the PBGT supporting tools as needed. Testers may use PBGT to craft test models that fulfill the desired testing goals for a given application. The developers may extend the tool with additional functionalities, for instance, defining new UITPs or implementing additional TC generation algorithms.

The PBGT process sets a total of 6 main steps (Figure 6): 2 fully automatic and 4 requiring additional manual effort: building model, configuration, mapping, TC generation, TC execution, and analyzing results.

The first step of the PBGT testing process is to build the test model. For that, the tester drags and drops the elements (Nodes) of the PARADIGM language available in the palette to the modeling area and connects them using the connectors (Links). The sequence established by the connections define the execution order of the testing strategies associated with each Node. Another way is to extract the model by a reverse engineering process and improve the model as needed from there.

After building the model, the next step is the configuration. In this step, the tester defines, for each Node within the model, the input data, preconditions, and checks to perform during test execution. This is the process that allows to adapt the generic test strategies within UITPs for testing slightly different implementations of a common behavior.

After configuration, it is possible to generate TCs. This is a fully automatic process that starts by generating all paths from the Init node to the End node and then generate TCs according to the configurations provided by the tester.

To execute the generated TCs over a Web application, it is necessary to establish a mapping between the model UI Test Patterns and the UI Patterns of the Web application to be tested. This mapping process is performed by a point and click process selecting each UITP within the model and relating each of those with the GUI controls in which the testing strategy will be executed. During this process the tool saves a screenshot of the control and information about its properties (for instance, ID and coordinates). This will allow the test execution module to know which web elements to interact with during TC execution. When Selenium fails to identify a GUI control based on its properties, Sikuli uses the saved image to identify it. For example, for mapping the Login UI Test Pattern within a certain test model, the user/tester must relate the login and password fields with the login and password textboxes of the SUT by clicking on them.

After the mapping step, it is possible to execute automatically the generated TCs over a Web application. During TC execution, the PARADIGM-TE reads the TCs and the mapping information gathered. For each step within the current TC being executed, it tries to identify the Web control to interact with based on its properties and image and then performs the testing actions over it. At the end, it builds a test execution report with the results achieved. When a failure is found, the tester needs to analyze if its origin is in the Web application or in the model and decide which updates are necessary to generate and execute TCs again. If the results obtained are satisfactory to the tester (eg, no more failures are found), the process ends. In previous work, we demonstrate how PBGT may be used to test real Web applications [6].

4 | USER INTERFACE TEST PATTERNS

Graphical user interfaces regularly feature UI Patterns. User interface patterns are graphical and interaction standards of a given functionality. User interface patterns can be realized using several different implementations, but they have common behavior. User interface test

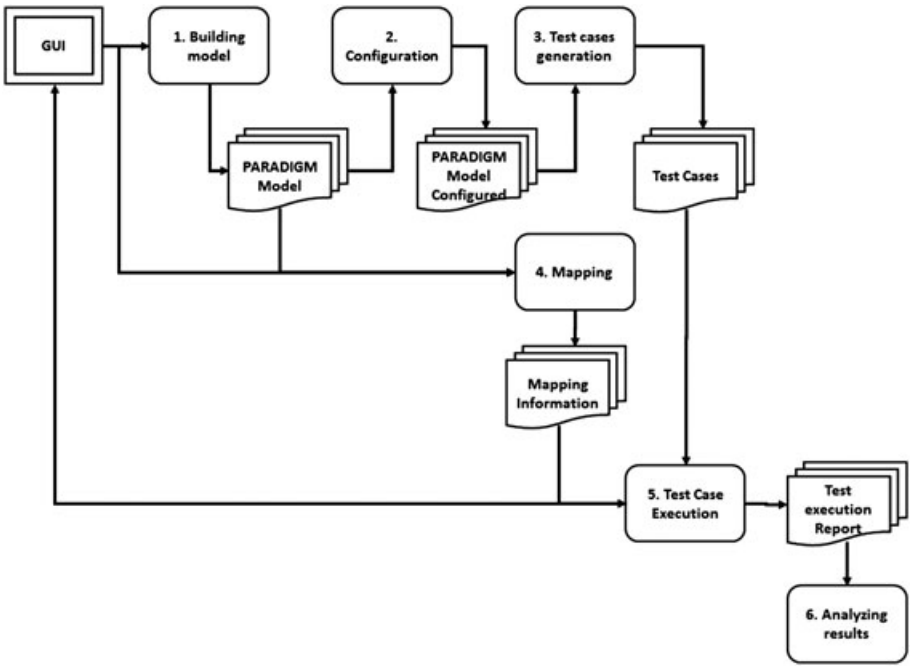

FIGURE 6 PBGT Pattern based GUI testing process

patterns provide a generic test strategy to test these different implementations. These patterns correspond to the Behavioral elements (Figure 4) of the PARADIGM language.

4.1 | Pattern language

Definition 1. A UI Test Pattern describes a generic test strategy, formally defined by a set of test goals, for later configuration, denoted as $< Goal, V, A, C, P >$ where

1. ***Goal*** is the ID of the test;
2. ***V*** is a set of pairs {[variable, inputData]} relating test input data (defined by the tester) with the variables involved in the test;
3. ***A*** is the sequence of actions to perform during TC execution. These actions are performed automatically over the application under test simulating user actions, eg, click on a button and send text to a text box;
4. ***C*** is the set of possible checks to perform after the sequence of actions (defined in *A* with input data defined in the variables in *V*) have been executed; and
5. ***P*** is a Boolean expression (precondition) defining the set of states in which it is possible to execute the test.

Commonly, *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* the test strategy 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 configures each UI Test Pattern within the model by providing test input data, selecting the checks to perform, and defining the preconditions for each test goal within such UITP. The tester can select the same test Goal multiple times for a UI Test Pattern providing different configurations. This allows to define data to test, for instance, an authentication mechanism for different situations, such as blank login, blank password, or invalid login/password. At this moment, UITPs do not impose any particular testing technique, as the ones in [34], so the tester has the freedom to choose the one he wants.

After researching generic testing strategies for the most common UI Patterns, from various sources [35–40], we have designed a set of UI Test Patterns. The UI Test Patterns identified, until now, are denoted as Input, Login, Master/Detail, Find, Sort, Call, and Option Set. We name them Base UI Test Patterns.

Patterns can be described in several styles [41–44]. The adoption of a given style depends on the subject and desired purpose. The description of Base UI Test Patterns that are going to be introduced next has the goal to concentrate the information necessary to start using the patterns, *when* to use them, *how* to use them, *why* they should be used, and finally, *what* they should be used for. Therefore, we will present the patterns, having in mind simplicity and comprehension, according to the following structure, with guidance from [44]:

- **Pattern Name**: unique identifier to shortly refer the pattern;
- **Context**: situation where the problem occurs;
- **Problem**: description of the problem addressed by the pattern;
- **Forces**: reflections to consider when choosing a solution to the problem;
- **Solution**: description of the proposed solution for the pattern;
- **Known Uses**: real conditions (UI Patterns) where the (UI Test) patterns can be applied;
- **Example**: concrete example of the applicability of the pattern.

Two motivations that can be seen as Forces for this pattern language, refer to the capability to have generic UI Test Patterns, to facilitate reuse, and significantly facilitate the task to identify when the UI Test Patterns, should be applied.

4.1.1 | Input UI Test Pattern

4.1.1.1 | Context:

Graphical user interfaces often feature input fields to gather data from the user. Input can be obtained via different ways, depending on the implementation, for instance, dialog windows, textboxes, drop-down menus, and window prompts.

4.1.1.2 | Problem:

How to define a test strategy that can be reused for testing the input functionality over its different possible implementations?

Graphical user interfaces vary in implementation in validating input data. Some GUIs display a message when the input is not valid, while others do not allow invalid data input. For instance, in some situations, selecting dates can be performed using time pickers. In this case, it is not possible to have an invalid data since we are selecting 1 specific date (from a set of options) instead of having to write it manually. However, the tester can check both situations (check for valid and check for invalid data) or check if the error message is displayed. In generic terms, testers aim to verify the behavior of input fields, by providing valid and invalid data, to check how the GUI responds to such input variations. Since the input can be gathered in several ways, the tester should have a generic, unique, and reusable way to provide input data, to test the behavior of input fields, across their different implementations.

4.1.1.3 | Forces:

- Should facilitate checking the expected behavior for valid and invalid input data;
- Should provide an intuitive and simple configuration (for valid and invalid input data).

4.1.1.4 | Solution:

Provide a generic test strategy for GUIs that feature input fields or drop-down menus, across their different implementations. The test strategy consists in the following:

1. Test **Goals**: "Valid data" (INP_VD) and "Invalid data" (INP_ID);
2. Set of variables **V**: {input};
3. Sequence of actions **A**: [provide *input*].
4. Set of checks **C**: {"message box X", "label Y", "error provider Z"} where X, Y, and Z correspond to the text to be displayed;

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.

4.1.1.5 | Known Uses:

- Input Prompt [35, 37, 38];
- Forgiving Format [35, 38];
- Input Controls [37].

4.1.1.6 | Example:

A tester aims to verify the behavior of input fields (Figure 7) for valid and invalid input data. In this case, with the Input UI Test Pattern, the tester is able to verify the correctness of the GUI.

Possible configurations for the Input UI Test Pattern could be

Goal– Valid data
V : { [Probability, 0.5]};
A : [Provide *Probability*];
C : {};
P : True.

In this configuration, the tester provides a valid input, since the probability is expressed within the range 0 to 1, and the tester provides value 0.5.

Goal– Invalid data
V : {[Probability, 1.4]};
A : [Provide *Probability*];
C : {Label "ERROR: Probability must be between 0 and 1."};
P : True.

The tester wants to verify if an error is displayed (in a label above the textbox) when providing invalid data – value 1.4 for the probability field.

4.1.2 | Login UI Test Pattern

4.1.2.1 | Context:

Numerous software systems have restricted functionalities that are only available after a successful authentication. Typically, users have to provide a username and a password.

4.1.2.2 | Problem:

How to define a test strategy that can be reused for testing authentication functionality over its different possible implementations?

Testers target to verify the behavior of the authentication functionality, where the objective is to check if it is possible to authenticate in the system with a valid username/password and check if it is not possible to authenticate otherwise.

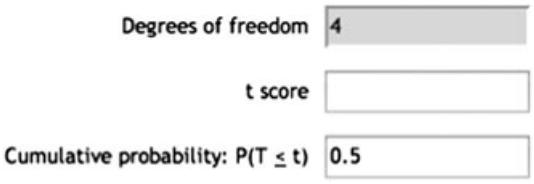

FIGURE 7 User interface part featuring the input functionality from Stattrek—Online Statistical Table [45]

4.1.2.3 | Forces:

- Should facilitate checking the expected behavior for valid login and invalid login;
- Should provide an intuitive and simple configuration (for valid and invalid login).

4.1.2.4 | Solution:

Provide a generic test strategy for GUIs that feature authentication mechanisms. The test strategy consists in the following:

1. Test **Goals**: "Valid login" (LG_VAL) and "Invalid login" (LG_INV);
2. Set of variables **V**: {username, password};
3. Sequence of actions **A**: [provide *username*; provide *password*; press *submit*];
4. Set of checks **C**: {"change to page X", "pop-up error Y", "same page", "label with message K"}, where X is the target page, Y and K the message to be displayed.

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.

4.1.2.5 | Known Uses:

Log In [38] (referred as *Login* in [46]).

4.1.2.6 | Example:

A tester aims to verify the functional correctness of the GUI that features an authentication mechanism (Log In UI Pattern as displayed in Figure 8). In this case, with the Login UI Test Pattern the tester is able to verify the correctness of the GUI. Thus, the testing goals are test the authentication for (1) valid and (2) invalid Username/Password.

Possible configurations for the Login UI Test Pattern could be

Goal– Invalid data
V : { [Email or Phone, "RodrigoPatternsPT@gmail.com"], [Password, "pattern56532"]};
A : [Provide *Email or Phone*, Provide *Password*, Press *Log In*];
C : {Change to page "Welcome"};
P : True.

In this configuration, the tester provides a valid username and valid password. He wants to verify if upon successful authentication, the application redirects the user to another page.

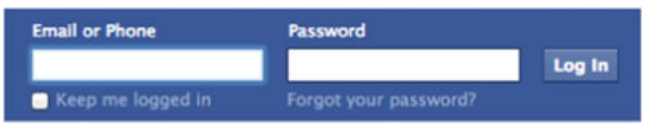

FIGURE 8 User interface part featuring the authentication functionality from Facebook [47]

Goal– Invalid data

V : {[Email or Phone, "RodrigoPatternsPT@gmail.com"], [Password, "test"]};

A : [Provide *Email or Phone*, Provide *Password*, Press *Log In*];

C : {Pop-up error "Please re-enter your password. The password you entered is incorrect. Please try again (make sure your caps lock is off)."};

P : True.

This configuration is different from the previous one, since the tester provides a valid username but an invalid password. He desires to verify the behavior for an invalid login, ie, if the user is redirected to a different page upon unsuccessful login.

4.1.3 | Master/Detail UI Test Pattern

4.1.3.1 | Context:

Depending on the situation, GUIs have a set of items to be displayed (they represent an area called *master*), where each item has associated content (detail). The detail is only visible upon selection of items from the master.

4.1.3.2 | Problem:

How to define a test strategy that can be reused for testing Master/Detail behavior over its different possible implementations?

Testers want to verify the behavior of 2 related objects in a GUI (master and detail), where the idea is to check if changing the master's value correctly updates the contents of the detail.

4.1.3.3 | Forces:

- Should facilitate checking the expected behavior when changing the master;
- Should provide an intuitive and simple configuration (for change master).

4.1.3.4 | Solution:

Provide a generic test strategy for GUIs that feature Master/Detail implementations (2 related objects). The test strategy consists in the following:

1. Test **Goal**: "Change master" (MD);
2. Set of variables **V**: {master, detail};
3. Sequence of actions **A**: [select *[master]*];
4. Set of checks **C**: {"detail has values *X*", "detail does not have values *X*", "detail is empty", "detail has *N* elements"}.

During configuration, the user has to provide master input data for MD, select the checks to perform, and define the precondition.

4.1.3.5 | Known Uses:

- Two-panel selector [35];
- Breadcrumbs [35, 37, 38, 46];
- Tab Bars [46] (referred as *Tabs* in [36]; referred as *Navigation Tabs* in [38]);
- Sequence Map [35];
- Accordion [35, 36, 38];
- Collapsible Panels [39];
- Details on Demand [39].

4.1.3.6 | Example:

A tester aims to verify the functional correctness of the GUI that features a Master/Detail area (Figure 9). The tester will use the Master/Detail UI Test Pattern to verify the correctness of the GUI. The aim is to check if changing the value of Make, the Model set of values changes accordingly.

One possible configuration that the tester could do

Goal– Change Master

V : { [Make, "Aston Martin"], [Model, "DB9"]};

A : [Select *Aston Martin*];

C : {"detail has value DB9"};

P : True.

For the above configuration, the result would be evaluated to True, since DB9 is 1 model from Aston Martin.

4.1.4 | Find UI Test Pattern

4.1.4.1 | Context:

Graphical user interfaces provide search engines to find information. Some implementations show the result obtained while the user is typing. Others present the result only at the end of the input after submission.

4.1.4.1 | Problem:

How to define a test strategy that can be reused for testing the search functionality over its different possible implementations?

Testers target to verify the behavior of the search functionality, to check if the result of a search is as expected, ie, if it find the right set of values.

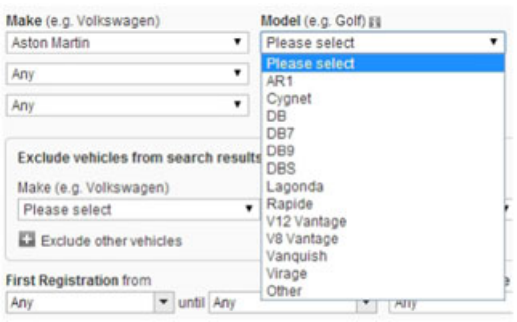

FIGURE 9 User interface part of Mobile.de[48] that features the Master (Make)/Detail (Model) area

4.1.4.2 | Forces:

- Should facilitate checking the expected behavior for value found and value not found;
- Should provide an intuitive and simple configuration (for value found and value not found).

4.1.4.3 | Solution:

Provide a generic test strategy for GUIs that feature search functionalities. The test strategy consists in the following:

1. Test **Goals**: "Value found" (FND_VF) and "Value not found" (FND_NF);
2. Set of variables **V**: $\{v_1, \ldots, v_N\}$, where *N* is defined during configuration time by the user/tester;
3. Sequence of actions **A**: [provide v_1, … ,provide v_N];
4. Set of checks **C**: {"result is an empty set", "result has *X* elements", "results does not have element *X*", "result has element *X* in line *Y*", "results have more than *X* elements", "results have less than *X* elements"}.

During configuration, the user has to define the value for *N*, provide input data for variables $v_1, \ldots, v_N$, select the checks to perform, and define the precondition.

4.1.4.4 | Known Uses:

- Auto Complete [37–39] (referred as *Auto Completion*in [35]);
- Search [35, 46];
- Fill In The Blanks [35, 38];
- Table Filter [38];
- Advanced Search [39];
- Search Box [39];
- Search Area [39].

4.1.4.5 | Example:

A tester aims to verify the functional correctness of the GUI that features a search box (Figure 10). The tester will use the Find UI Test Pattern to verify the correctness of the GUI. He aims to check if it is possible to find the right set of news in the news' archive. Hence, the testing goals are test the search for (1) value found and (2) value not found.

Possible configurations that the tester could perform:

Goal– Value found

V : { [SearchFor, "Ferrari"] };
A : [Provide *data*];
C : {"result has 12103 elements"};
P : True.

FIGURE 10 User interface part of GPUpdate search news functionality[49]

With this configuration, the tester aims to check if there are 12 103 news related with "Ferrari" cars. The test passes because it is exactly the number of news obtained.

Goal– Value not found

V : { [SearchFor, "Monaco"] };
A : [Provide *data*];
C : {"result has 3412 elements"};
P : True.

In this configuration, the tester wants to verify if there are 3412 news related with "Monaco." The test fails because the result set has 3277 news.

4.1.5 | Sort UI Test Pattern

4.1.5.1 | Context:

When displaying a set of results, the user is able to sort them according to a given criteria (ascending or descending).

4.1.5.2 | Problem:

How to define a test strategy that can be reused for testing sort functionality over its different possible implementations?

Testers target to verify the behavior of the sort functionality, to check if the result (of a sort action) is ordered accordingly to the chosen sort criterion. In some implementations, the ordering is only performed over a specific field. Yet other implementations allow to define several fields for ordering.

4.1.5.3 | Forces:

- Should facilitate checking the expected behavior for sort actions (ascending and descending);
- Should provide an intuitive and simple configuration (according to given criteria).

4.1.5.4 | Solution:

Provide a generic test strategy for GUIs that feature sort mechanisms. The test strategy consists in the following:

1. Test **Goals**: "ascending" (SRT_ASC) and "descending" (SRT_DESC);
2. Set of variables **V**: $\{(v_1, c_1), \ldots, (v_N, c_N)\}$, where *N* is defined by the user/tester during configuration time and "c_i" represents the criteria defined for the given variable "v_i";
3. Sequence of actions **A**: [provide (v_1, c_1), … , provide (v_N, c_1)];
4. Set of checks **C**: {"element from field *X* in position *Y* has value *Z*", "elements (v) with a given criteria (c) are sorted accordingly", "element with value *X* is before element with value *Y*}.

During configuration, the user has to define the value for *N*, provide input data for the pair variables and criteria $(v_1, c_1), \ldots, (v_N, c_N)$, select the checks to perform, and define the precondition.

4.1.5.5 | Known Uses:

- Sort by Column [38] (referred as Sortable Columns in [40]);
- Table Sorter [41, 39] (referred as Sortable Table in [35]).

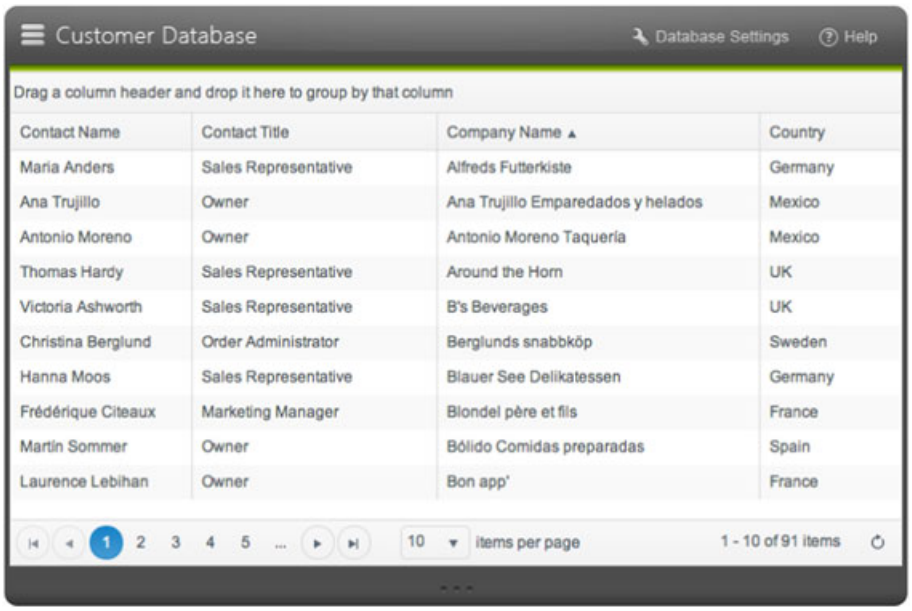

FIGURE 11 User interface part of Telerik[50] that features a sort functionality

4.1.5.6 | Example:

A tester aims to verify the functional correctness of the GUI that features a sort functionality (Figure 11). The tester will use the Sort UI Test Pattern to verify the correctness of the GUI. Thus, the testing goals are "Ascending" and "Descending."

One possible configuration that the tester could do

Goal– Ascending

V : { ("Company Name", ascending) };
A : [Provide (*Company Name*, *ascending*)];
C : {"element from field *Company Name* in position *4* has value *Alfreds Futterkiste*"};
P : True.

The previous configuration returns the result False. When ascending the column "Company Name" the value "Alfreds Futterkiste" does not appear in position 4.

Goal– Descending

V : { ("Company Name", descending)};
A : [Provide (*Company Name*, *ascending*)];
C : {"element from field *Company Name* in position *2* has value *Alfreds Futterkiste*"};
P : True.

In this configuration, the goal is to verify if elements are ordered in descending criteria. The value "Alfreds Futterkiste" does not appear in position 2. This configuration is evaluated to False.

4.1.6 | Call UI Test Pattern

4.1.6.1 | Context:

Users trigger actions in GUIs. For instance, to register an account, users have to provide input and then, to complete the process, they have to submit the data to the system (press submit button).

FIGURE 12 User interface part of Mobile.de [48] that features an action control

4.1.6.2 | Problem:

How to define a test strategy that can be reused for testing the invocation/call functionality over its different possible implementations?

Testers aim to check the functionality of a corresponding invocation.

4.1.6.3 | Forces:

- Should facilitate checking the expected behavior for a call action;
- Should provide an intuitive and simple configuration (call succeeded and call failed).

4.1.6.4 | Solution:

The test strategy consists in the following:

1. Test **Goal**: "Action invoked" (CL_AS);
2. Set of variables **V**: {};
3. Sequence of actions **A**: [press];
4. Set of checks **C**: {"pop-up message", "stay in the same page", "change to page *X*"}.

During configuration, the user has only to select the checks to perform and define the precondition.

4.1.6.5 | Known Uses:

Due to the *atomic* nature of this UI Test Pattern, it can be applied practically everywhere.

FIGURE 14 PARADIGM model of a simple Web application

4.1.6.6 | Example:

A tester aims to verify the functional correctness of the GUI, by checking the behavior of pressing the *Show results* button (Figure 12). The tester will use the Call UI Test Pattern to verify the correctness of the GUI. Thus, the testing goal is "Action invoked."

One possible configuration for this UI Test Pattern could be

Goal– Action Invoked
V : { };
A : [press];
C : {"change to page My car search"};
P : True.

In this configuration, the tester wants to verify if upon action invoked, causes a change to a different page.

4.1.7 | Option Set UI Test Pattern

4.1.7.1 | Context:

Given a set of choices, the users may select multiple elements in a GUI.

4.1.7.2 | Problem:

How to define a test strategy that can be reused for testing the multiple selection functionality over its different possible implementations?

Testers aim to check the functionality corresponding to multiple selections.

4.1.7.3 | Forces:

— Should facilitate checking the expected behavior for multiple selections (selection succeeded and selection failed);
— Should provide an intuitive and simple configuration (for selection succeeded and selection failed).

4.1.7.4 | Solution:

The test strategy consists in the following:

1. Test **Goal**: "Selection invoked" (OST_OK);
2. Set of variables **V**: $\{v_1, \ldots, v_K\}$ corresponding to the set of possible selections;
3. Sequence of actions **A**: [select v_i, … , select v_j], where i, … ,j belong to [1, … ,K];
4. Set of checks **C**: {"pop-up message", "stay in the same page", "change to page *X*"}.

During configuration, the user has to define the number of options (k), provide the k possible values, identify the selections, select the checks to perform, and define precondition.

4.1.7.5 | Known Uses:

— Live Filter [38];
— Array of N checkboxes [35];
— Array of N toggle buttons [35];
— Multiple-selection list or table [35];
— List with checkbox items [35].

4.1.7.6 | Example:

The tester aims to check the behavior after selecting a subset of the options displayed in Figure 13. The tester will use the Option Set UI Test Pattern to verify the correctness of the GUI. Thus, the testing goal is "Selection invoked."

One possible configuration for the Option Set UI Test Pattern could be

Goal– Selection Invoked
V : { Auxiliary heating, Bluetooth, CD player, Central locking, Cruise control, Electric windows, Hands-free kit, Head-up display, isofix (child seat anchor points), MP3 interface};
A : [select *CD player*, select *Cruise control*, select *Electric windows*, press *Show results*];
C : {stay in the same page};
P : True.

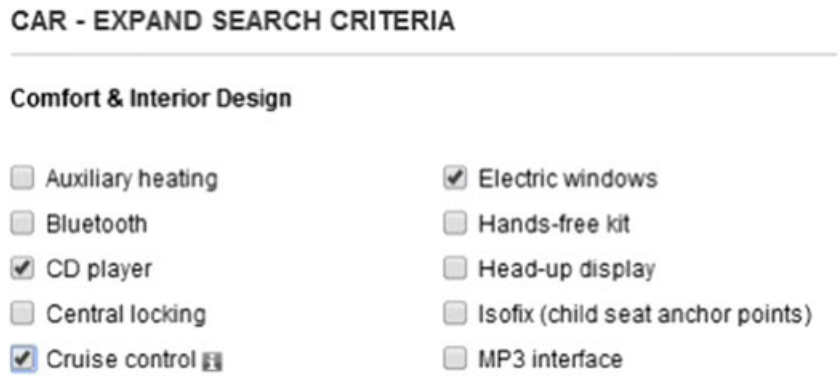

FIGURE 13 User interface part of Mobile.de [48] that features a control with multiple options

In this configuration, the tester selects CD player, Cruise control, and Electric windows. He wants to verify if the application remains in the same page. This configuration will be evaluated to True, since the application remains in the same page.

4.2 | Application test model

Within PBGT approach, the models are written in PARADIGM and configured by testers with the support of the PARADIGM-ME tool or by providing an XML file with all the configurations needed for the model. PARADIGM models are written in accordance with the testing goals defined for the applications under test.

A simplified model of a Web application can be found in Figure 14. The model contains 3 elements. The first is a Login UI Test Pattern (element 1) with a valid and an invalid configuration (Figure 15). If the login credentials are valid, the Web application changes page and the user can then perform a search (element 2, Find UI Test Pattern). If the login credentials are invalid, the Web application stays on the same page. The user can also go to the page "Contacts" (element 3, Call UI Test Pattern) without passing through the Login UI Test Pattern.

4.3 | Test case generation

Generation of TCs from models is a challenge because it incorporates a significant impact on the effectiveness of the overall test. In the context of PBGT, TCs are generated automatically from a PARADIGM model. Generically, the first step is to flatten the model into 1 level by recursively expanding all structural (*Forms* and *Groups*) elements. After this process, the model does not have *Forms* nor *Groups*, so every node is a UI Test Pattern. Then, the tool generates (whenever possible) all possible paths that traverse the model from *Init* to *End*. In short, a test path is a sequence of UI Test Patterns. For instance, for the model in Figure 14, the test paths are [1,2] and [3], where 1, 2, and 3 are the IDs of the UI Test Patterns within the model in Figure 14.

However, there are some peculiarities within PARADIGM models that should be considered. Elements within a PARADIGM model can be optional. In that case, the tool will generate test paths that traverse that element and others that do not traverse such element. In addition, elements may be inside *Groups*. In that case, it means that those elements can be executed in any order, so the algorithm will generate different test paths that will cover different permutations of those elements For instance, a *Group* with "N" loose mandatory elements, will generate at most "N!" (N factorial) test paths [6].

Occasionally, depending on the testing goals and the model, the number of the test paths can be massive and some times even unfeasible. There will be situations where the tester will only want to work with a small set of test paths to pursue a given testing goal or even to deal with scalability issues. To tune the number of test paths generated, PARADIGM-ME provides a set of filters. For example, it is possible to define

- an upper limit for the number of test paths generated,
- an upper limit for the permutations of the elements inside a Group (elements that can be executed in any order), and
- in case the model (seen as a graph) has cycles, it is possible to define the maximum number of times each cycle is traversed.

From 1 test path, it is possible to generate several TCs, depending on the configurations defined for the UI Test Patterns. One UI Test Pattern can contain multiple configurations (*UITPConf*). For example, a Login UI Test Pattern can have configurations to test valid and invalid authentications, and each of them can lead to different outcomes. Therefore, each configuration must generate a different TC. A TC is then a sequence of UI Test Patterns configurations (*UITPconf*). So consider a test path with 2 UI Test Patterns ([UITPi, UITPj]). If there are 2 configurations defined for UITPi (UITPConfi.1, UITPConfi.2) and 1 configuration defined for UITPj (UITPConfj.1), the algorithm will generate 2 TCs:

$$Tc1 = [UITPconf_{i.1}, UITPconfj.1], \tag{1}$$

$$Tc2 = [UITPconf_{i.2}, UITPconfj.1]. \tag{2}$$

So, considering a Test Path with x UI Test Patterns, each with y_x configurations, the total maximum number of generated TCs will be given by the following formula:

$$Tc = y_x \times y_{x-1} \times \ldots \times y_1. \tag{3}$$

Each UITP configuration has a precondition establishing the states where a test strategy can be run so, most of the times, the total number of TCs is less than the calculated by formula (3). For instance, consider that we have a Login UITP with 2 configurations: 1 for valid login and other for invalid login, followed by an input UITP with also 2 configurations both with preconditions saying that the corresponding test strategy would run only after valid logins. In this case, instead of 4 TCs, we would obtain only 3. However, even with preconditions, it is possible to construct models for which the number of generated TCs is so huge that it can compromise the generation and execution of TCs in feasible time. There are different ways to deal with this problem [51, 52], and it is possible to extend the PBGT tool with additional TC generation algorithms in the future. At this moment, to deal with scalability issues, the PARADIGM-ME tool provides some other TC generation algorithms and tune properties that generate a subset of the overall TCs as follows:

— **Default**. Generates only N TCs (the first N generated by the algorithm) per test path, where N is defined by the tester;

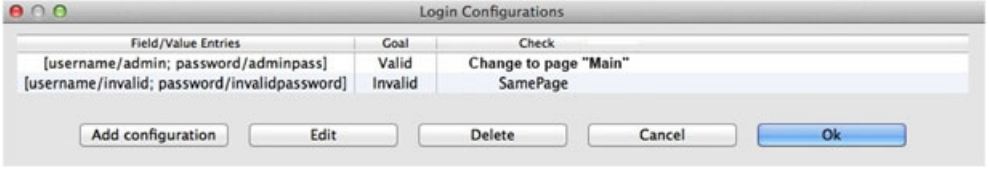

FIGURE 15 Configurations within PARADIGM-ME, for Login UI Test Pattern (number 1) from Figure 14

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Script>
<Path value="[1, 2]">
<Init/>
<Login flag="false" name="Login" number="1" optional="false" validity="Valid">
    <Check check="ChangePage" result=""/>
    <Value field="username_" fieldName="username" value="admin"/>
    <Value field="password_" fieldName="password" value="adminpass"/>
    <Button field="button_1"/>
</Login>
<Find flag="false" name="Search" number="2" optional="false">
    <Check check="NumberOfResults_more_than" position="1" result="50"/>
    <Value field="Name_2" fieldName="Name" value="Computer Science"/>
</Find>
</Path>
</Script>
```

FIGURE 16 Description of a test case in XML format

- **Invalid Configurations**. Generates only TCs that traverse invalid configurations, for example, it will contain TCs that check the behavior of invalid authentications regarding Login UI Test Pattern;
- **Random**. Generates TCs arbitrarily selecting the configurations of the UI Test Patterns within randomly selected test paths. The algorithm stops after a predetermined time interval.

In addition, to the TC generation algorithms and filters above, the tool allows to tune even further the TC generation algorithm to deal with scalability issues:

- **Mandatory and Exclusion Elements**. In case, the tester wants to test a certain feature or a certain workflow that can only be reached after crossing a certain path; he can explicitly define the mandatory and exclusion elements (nodes). By excluding 1 element (or a sequence of elements), every TC that contains that element will not be generated.
- **Specific UI Test Patterns configurations**. When the tester wants to select a subset of configurations to include in the generated TCs, for example, when only a part of the functionality of the application is implemented, all the other configurations will be ignored.
- **Number of TCs**. The tester defines the maximum number of TCs to generate from the model.

After selecting and tuning the TC generation algorithm, it is possible to generate TCs and execute them over the GUI under test.

The TCs generated are saved in a XML file. Considering the model presented in Figure 14, the TC generated for the test path [1, 2], which checks a valid authentication for the Login UI Test Patterns (element 1) followed by checking the number of results obtained by searching "Computer Science" for the Find UI Test Pattern would be as shown in Figure 16.

5 | EMPIRICAL EVALUATION

To assess the usefulness, applicability, and capability in finding failures with the PBGT approach, we have conducted 2 case studies following Yin's guidelines [53]. These guidelines cover all aspects of a case study research method. Thus, when conducting a case study, there are 6 major activities to be followed: (1) Plan—consists in identifying the research questions and idealize the case study; (2) Design—define data to be collected as well as procedures to maintain the case study quality (external validity and internal validity) and identify the criteria to interpret the findings; (3) Prepare—identifying teams to be part of the case study and define procedures for data collection; (4) Collect—case study execution with data gathering; (5) Analyze—consists in examining, categorizing, and tabulating data to draw empirically based conclusions; and (6) Share—report the case study demonstrating its findings and results.

5.1 | Case Study 1: assessing PBGT failure detection capability in fielded systems

This study aims to assess the feasibility of the PBGT approach. In particular, it aims to determine whether the testers are able to effectively use PBGT to detect failures and measuring the effort required to start using PBGT. This study is designed to answer the following research questions:

RQ1.1 What is the effort required to start using the PBGT approach?
RQ1.2 What is the proficiency of the PBGT approach in finding failures?

5.1.1 | Metrics

To address the research questions above, we have defined a set of metrics to be gathered during the execution of the study. The metric (**M1.1**) was related to the research question **RQ1.1** and kept track of the time required to present the PBGT approach, the PARADIGM language and the modeling environment to the teams, and the time the teams spent in building and configuring the necessary models to test the subject systems. The second metric (**M1.2**) related to research question **RQ1.2** and stored information about the number of failures that the teams were able to find in field applications.

5.1.2 | Subject systems

To prepare the case study, we selected a set of subject systems, according to the following criteria: (1) the subject systems were required to have a variety of functionalities, so they could provide the ability to apply and fulfill at large scale, the different PBGT testing strategies; and (2) because of the increasing usage of Web applications in recent days, we opted to choose public Web-based applications. Therefore, the selected subject systems were

- **SS1.1** – mobile.de [48]. This is a german marketplace for buying and selling new and used vehicles. It allows to search vehicles according

to various parameters (make, model, kilometers, year, etc), sort the search result according to a set of parameters, and refine the search. The website also allows registration for users who, after authentication, can insert vehicles for sale, among other functionalities.

- **SS1.2** – australian-charts [54]. This is an australian site that features the best selling music singles and albums in Australia. It displays 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, subsequently, authentication; and provides support for forums, among other functionalities. Moreover, this site has another functionality that allows access sites for the same purpose (listing the best-selling music) from other countries, such as, Finland, Germany, etc.

5.1.3 | Testing goals

For this study, we have defined a set of testing goals to cover a subset of the use cases of the Web sites, ie, SS1.1 and SS1.2. The defined testing goals for SS1.1 are described in Table 1, and the ones for SS1.2 are represented in Table 2. The tables feature the ID that will be used to refer to the testing goal in question and the description of the testing goal.

5.1.4 | Setup

To start the case study, we assigned 2 teams composed by students, who were not familiar with the PBGT approach, to embrace the role of testers. The first team (to be referred as **Team 1.1**) was assigned **SS1.1**. The second team (referred as **Team 1.2**) was assigned **SS1.2**. To answer **RQ1.1**, we registered the time required to craft and configure models. Concerning **RQ1.2**, we collected the number of failures detected by each team regarding each subject system.

To run the study, the students had to install PARADIGM-ME tool and afterwards they gained access to the subject systems to test. The procedure that was followed to execute this study is described next.

TABLE 1 Defined testing goals for *mobile.de* (SS1.1)

ID	Description
TG1.1	Test search for models of a particular automotive brand
TG1.2	Test if the number of the cars within the result set is correct and if it contains the elements it is supposed to have
TG1.3	Test the possibility to sort the results providing different criteria
TG1.4	Test the search by refining its contents
TG1.5	Test if the items selected to limit the search are transferred to the search criteria area and vice versa.
TG1.6	Test the relationship between make and model
TG1.7	Test if providing input in the adjust search affects the results
TG1.8	Test the limit search and verify if it changes the results set

5.1.5 | Procedure

The study started by providing a 2-hour training session about the PBGT approach and the PARADIGM DSL to the teams. Doubts were clarified and simple examples illustrating the modeling, configuration, and testing steps were presented. Then, the subject systems to test were introduced. After that, both teams received the same set of testing goals (Tables 1 and 2) from which they would build the testing models.

By using PARADIGM-ME tool, the teams started building the models to fulfill the testing goals. During the study execution, they recorded the time spent in building and configuring the models. Afterwards, they generated TCs and start executing the tests, in PARADIGM-ME. Then they analyzed the test results and recorded the failures found and thus concluding the case study.

5.1.6 | Results

When the study execution finished, we collected the models created by the teams, the configurations applied to the models, the generated TCs, and a report listing the failures found. Because of the high amount of data obtained from this study, we decided to show and explain only 1 of the models built by 1 of the team members, as well as its configuration, the generated TCs, and failures found. However, all models, their configurations, generated test cases, and reports are documented in a public repository that was created to store the data that was collected from this study. The public repository can be found in [55].

An example of a model (consisting of 2 levels) created by 1 member from Team 1.1 regarding SS1.2 is illustrated in Figure 17.

The tester created the model from Figure 17 to achieve the testing goals from Table 2. In SS1.1, the end user is able, without any forced order, to register, login, search for songs/albums/artists, select countries according to his preference, and select several elements from the left menu (referred as Options for now on). Therefore, in PARADIGM, this arbitrary access to functionality is described by the *Group* element (labeled Actions, element number 1). This *Group* element is comprised by several elements: a Login UI Test Pattern (labeled "Login," element number 1.3) that the tester selected to check TG1.9; a *Form* element called "Register" (element number 1.1) to check TG1.10 to test the user registration; another *Form* element that refers to

TABLE 2 Defined testing goals for *australian-charts.com* (SS1.2)

ID	Description
TG1.9	Test the authentication functionality in several moments, for instance, after performing a search
TG1.10	Test new user registration
TG1.11	Test the dependency between the selection of a different country and the contents of a list with different options
TG1.12	Test the dependency between the selection of an element within a list of options and the content displayed in the middle panel
TG1.13	Test the simple search
TG1.14	Test the extended search
TG1.15	Test the top search

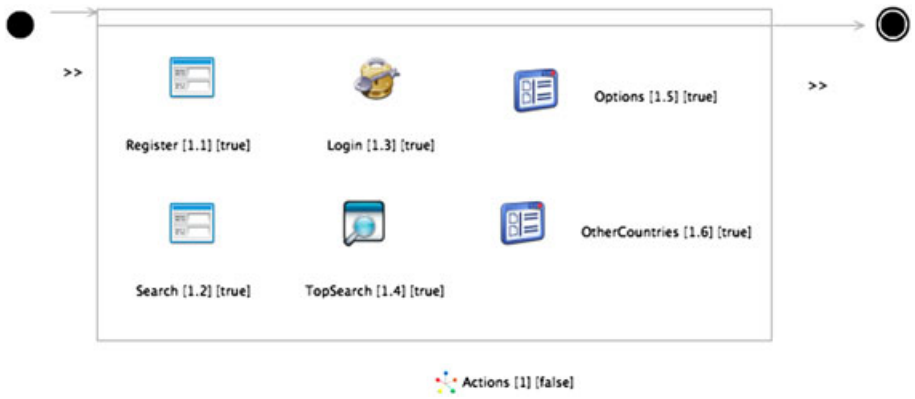

FIGURE 17 Australian-charts model (first level) written in PARADIGM

TABLE 3 Configuration details for UI Test Patterns featured in the model from Figure 17

Name	Type	Validity	Variables	Check
Login	Login	Valid	{[username, "pbgt"]; [password, "test"]}	{Change to page "Main"}
		Invalid	{[username, "pbgt"]; [password, "testtest"]}	{Label "Login failed. Unknown user or wrong password (…)"}
OtherCountries	MD	NA	{[Master, "United Kingdom"], [Detail, ["UK Charts", "Search"]] }	{Detail has 2 elements}
Options	MD	NA	{[Master, "Top 50 singles"], [Detail, "Of Monsters and Men – Little Talks"]}	{Detail has "Of Monsters and Men – Little Talks" element}
TopSearch	Find	Valid	{ [SearchForum, ""] }	{Result has element "show all topics" in line 1}
		Invalid	{[SearchForum, "metallica"] }	{Result is an empty set}

Abbreviation: UI, user interface.

several searches (element number 1.2, labeled "Search"); a Find UI Test Pattern (labeled "TopSearch," element number 1.4) to address TG1.15; and 2 Master/Detail UI Test Patterns, where the first element labeled "Options" number 1.5 fulfills TG1.12 and the second element labeled "OtherCountries" number 1.6 addresses TG1.11.

The configuration performed for the above model is represented in Table 3. This table contains the details in respect to the configuration for the UI Test Patterns that are featured in the model. It indicates the name of the element referred in the model, the type of the UI Test Pattern and the different configurations (valid and invalid), variables to be used, and the checks performed for each UI Test Pattern featured in the model. Each row contains the configuration details for each UI Test Pattern from Figure 17. For instance, the first row displays the configuration for the Login UI Test Pattern (element number 1.3 in Figure 17). The definition for the Login UI Test Pattern is described in Section 4.1.2. Therefore, according to the pattern definition, the tester needs to provide valid username/password for LG_VAL (login valid) and invalid username/password for LG_INV (login invalid) and select the checks to perform. In this case, for LG_VAL, the tester defined "pbgt" for the username, "test" for the password and the selected check was to verify if the user changes to page "Main." For LG_INV, the tester chose "pbgt" for the username but provided an invalid password, ie, "testtest." The selected check for LG_INV was to verify if upon unsuccessful login the system would display in a label the "Login failed. Unknown user or wrong password (…)" message.

The model from Figure 17 contains a *Form* element that refers to the registration functionality. The contents of this *Form* are displayed in Figure 18. The registration *Form* contains a Call UI Test Pattern (element number 1.1.1 labeled "RegisterButton") that will trigger the access to the registration fields. These fields are Input UI Test Patterns, allowing the user to provide data such as username (element number 1.1.2.1 labeled "User"), password (element number 1.1.2.2 labeled "Password"), confirm password (element number 1.1.2.3 labeled "CPass"), and email (element number 1.1.2.4 labeled "Email"). All these elements are structured within a *Group* element (number 1.1.2 labeled "RegisterFields"), since they can be filled in arbitrary order. After providing the necessary data to make a registration, the user needs to press the "SignOn" button, to submit the entered data. The latter is modeled using a Call UI Test Pattern (element number 1.1.3 labeled "SignOn").

Table 4 holds the configuration of the elements (UI Test Patterns) from the model from Figure 18. For example, the configuration for the Input UI Test Pattern that refers to element number 1.1.2.1 labeled "User" from Figure 18 is displayed in the second row of this table. According to the pattern definition (in Section 4.1.1), the tester needs to provide valid input data (INP_VD) and invalid input data (INP_ID)

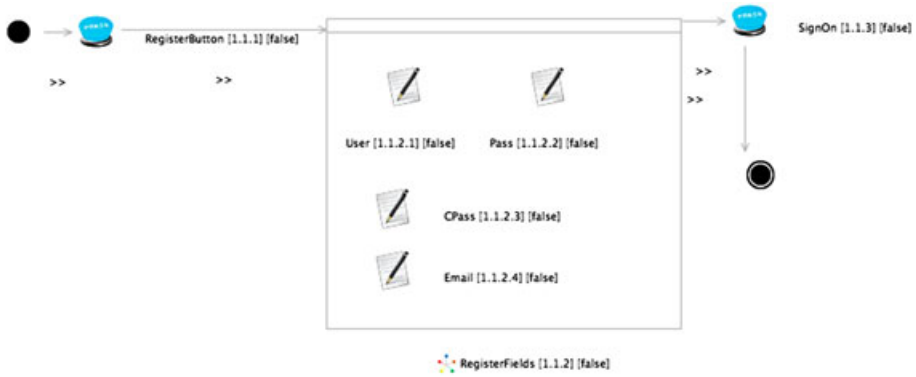

FIGURE 18 Australian-charts featuring the modeling of the registration form

TABLE 4 Configuration details for UI Test Patterns featured in registration form from Figure 18

Name	Type	Validity	Variables	Check
RegisterButton	Call	NA	{}	{Change to Page "Sign On"}
User	Input	Valid	{[Username, "PBGT"]}	{}
		Invalid	{[Username, ""]	{label "Please fill out all mandatory fields: Username"}
Password	Input	Valid	{[Password, "PBGTPass"]}	{}
		Invalid	{[Username, ""]}	{label "Please fill out all mandatory fields: Password"}
CPass	Input	Valid	{[ConfirmPassword, "PBGTPass"]}	{}
		Invalid	{[Password, ""]}	{label "Please fill out all mandatory fields: Confirm Password"}
Email	Input	Valid	{[e-Mail, "PBGT@gmail.com"]}	{}
		Invalid	{[e-Mail, "PBGTmail"]}	{label "An error occurred on the server when processing the URL. Please contact the system administrator"}
SignOn	Call	NA	{}	{Change to Page "Main"}

Abbreviation: UI, user interface.

and select the checks to perform. For INP_VD, the tester provided "PBGT" as the input for the username field without any check to be performed. Yet, for INP_ID, he chose a blank username and the check refers to a label containing the text "Please fill out all mandatory fields: Username."

The model in Figure 17 contains a second *Form* (element 1.2, labeled "Search") that refers to the advanced search functionality. The contents of this *Form* are illustrated in Figure 19. This *Form* contains a SearchButton, modeled as a Call UI Test Pattern, that will provide access to the simple and extended search functionalities. The simple (element number 1.2.2.1 labeled "SimpleSearch" and extended (element number 1.2.2.2 labeled "ExtendedSearch") searches are modeled as Find UI Test Patterns. They are structured within a *Group* (element number 1.2.2 labeled "SearchTypes"), since they can be used arbitrarily.

The configuration for the model from Figure 19 is displayed in Table 5. This table describes the configuration for the UI Test Patterns (one Call and 2 Find UI Test Patterns) featured in the *Form*. Considering the configuration for the Call UI Test Pattern (first row in Table 5), and according to the pattern definition (in Section 4.1.6), the tester has to select the checks to perform. For this element (number 1.2.1 labeled "SearchButton"), the tester selected the check "Change to Page Search," which means that he wants to verify that upon pressing the Search button, the system will redirect the user to another page (Page Search).

In PBGT, TCs are automatically generated from PARADIGM models. Because of high number of TCs, we only display the ones that led to failures found. Table 6 displays part of the generated TCs from the model from Figure 17. It contains the ID that refers to the identifier of the TC in question, the test path, and the TC described in XML format. The first row in this table refers to TC 1.1 (TC1.1). The path that led to the failure was 1.3 (refers to element number 1.3 in Figure 17, Login UI Test Pattern) then 1.4 (refers to element number 1.4—Find UI Test Pattern), and 1.6 (refers to element number 1.6—Master/Detail UI Test Pattern). The last column displays the TC in XML format. It contains all the configurations for the elements within the path (second column) that led to the failure.

5.1.7 | Findings

All the teams were able to build the model and fulfilled the expected testing goals. We validated and analyzed manually the models created by all members of the teams and compared the models between their owns. We concluded that the models are similar, only differing in their structure (depth levels). Some used *Forms* to feature further detail while other members opt to display all elements in 1 level.

FIGURE 19 Australian-charts featuring the modeling of the search form

TABLE 5 Configuration details for UI Test Patterns featured in Search Form from Figure 19

Name	Type	Validity	Variables	Check
SearchButton	Call	NA	{}	{Change to Page "Search"}
SimpleSearch	Find	Valid	{[SearchSong, "nothing"]}	{result has element "Alesha Dixon" in line 1}
		Invalid	{[SearchSong, "nothing"]}	{result has element "Sinead O' Connor" in line 5}
ExtendedSearch	Find	Valid	{[SearchAlbum, "justice for all"]}	{result has element "Metallica" in line 1}
		Invalid	{[SearchAlbum, "justice for all"]}	{result has element "Metallica" in line 25}

Abbreviation: UI, user interface.

We also measured the time required to create and configure the models to fulfill the testing goals. Moreover, we wanted to know PBGT capability in finding failures. Thus, the results of this case study are represented in Table 7 (**RQ1.1** and **RQ1.2**).

One failure was reported in SS1.1. This failure was related with the difference in the number of elements displayed in 2 different pages, after performing a search on the first page to later be submitted to the second page. The model and its configuration and the generated TCs can be found in [55].

A total of 4 failures were found in SS1.2. The first failure was caused by selecting "Top 50 Singles" and afterwards performing an invalid login (TC1.3). This resulted in the display of an empty white page. The second failure was triggered after performing a top search looking for forum and afterwards performing an invalid login (TC1.2). This displayed an error message: "Microsoft OLE DB Provider for SQL Server error...". The third failure was also related with the sequence of actions performed in the UI, namely, 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 (TC1.4). Finally, in the context of the latter, changing the country to "United Kingdom" does not show "Search" neither "UK Charts" in the related area as what happens with other countries (TC1.1).

In summary, the failures encountered in this case study were related with (1) sequence of actions performed in the UI, such as selecting elements after providing invalid logins; and (2) with dependencies between related UI elements.

Since the source code of these subject systems are not available (these are public Web sites not developed by the authors), it is not possible to establish a relation between failures found and the faults that were in its origin. But, regarding SS1.2, it seems that 3 of the 4 failures may be related to the same fault.

The teams reported some complaints regarding the usability of the modeling environment. It was directed towards the auto-alignment and placement of the connectors between elements, since it was not perfect, in visual terms, which required extra effort to correct. Furthermore, all teams agreed that the time spent in training (2 h) was sufficient to understand the PBGT approach and how to use PARADIGM-ME to model applications and start finding failures.

5.1.8 | Threats to validity

The limited number of subject systems in our evaluation can be seen as *external threat to validity*. However, to minimize this problem, we have used various public Web sites ranging in complexity and functionality.

For *internal validity*, the threat is related with the measurement concerning the modeling and configuration efforts. We leveraged all teams, to start from scratch in the way that they were not familiarized and did not have any experience with PBGT. If the study was repeated but to test other subject systems, the efforts concerning modeling and configuration would probably revealed lower, since the teams would already have some experience with PBGT.

5.2 | Case study 2: comparison among PBGT, manual model-based TC generation, and random testing

The purpose of the second case study is to compare the PBGT MBT approach with random testing and with the manual model-based construction of TCs regarding their ability to find failures and testing time spent on activities prior to performing the tests. The research questions designed for this study were

RQ2.1 How many failures are detected by the different PBGT TC generation techniques, by manual model-based TC construction and by random testing?

RQ2.2 What are the manual efforts required by PBGT and manual model-based TC generation?

TABLE 6 Part of the generated test cases from the model from Figure 17

ID	Test path	XML test case
TC1.1	[1.3, 1.4, 1.6]	<Script> <Path value="[1.3, 1.4, 1.6]"> <Init/> <Login flag="false" name="Login" number="1.3" optional="true" validity="Valid"> <Check check="change to page" result="main"/> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="test"/> <Button field="button_1.3"/></Login> <Find flag="false" name="TopSearch" number="1.4" optional="true"> <Check check="Result has element" position="1" result="show all topics" /> <Value field="" fieldName="searchforum" value="" /> </Find> <MasterDetail flag="false" name="OtherCountries" number="1.6" optional="true"> <Check N="2" check="ContainsN" /> <Master field="master_1.6" value="United Kingdom"/> <Detail field="detail_1.6"> <Value name="UK Charts"/> <Value name="Search"/> </Detail> </MasterDetail> </Path> </Script>
TC1.2	[1.4, 1.3]	<Script> <Path value="[1.4, 1.3]"> <Init/> <Find flag="false" name="TopSearch" number="1.4" optional="true"> <Check check="Result has element" position="1" result="show all topics" /> <Value field="" fieldName="searchforum" value="" /> </Find> <Login flag="false" name="Login" number="1.3" optional="true" validity="Invalid"> <Check check="Label" result="login failed. Unknown user or wrong password(_)"/> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="testtest"/> <Button field="button_1.3"/></Login> </Path> </Script>
TC1.3	[1.5, 1.3]	<Script> <Path value="[1.5, 1.3]"> <Init/> <MasterDetail flag="false" name="Options" number="1.5" optional="true"> <Check N="2" check="ContainsN" /> <Master field="master_1.5" value="Top 50 Singles"/> <Detail field="detail_1.5" Value="Of Monsters and Men - Little Talks"/> </MasterDetail> <Login flag="false" name="Login" number="1.3" optional="true" validity="Invalid"> <Check check="Label" result="login failed. Unknown user or wrong password(_)"/> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="testtest"/> <Button field="button_1.3"/></Login> </Path> </Script>
TC1.4	[1.2.1, 1.2.2.1, 1.3]	<Script> <Path value="[1.2.1, 1.2.2.1, 1.3]"> <Init/> <Call flag="true" name="SearchButton" number="1.2.1" optional="false"> <Check check="change to page" result="search"/> <Field name="call_1.2.1"/> </Call> <Find flag="false" name="Simplesearch" number="1.2.2.1" optional="true"> <Check check="Result has element" position="1" result="Alesha Dixon" /> <Value field="" fieldName="SearchSong" value="nothing" /> </Find> <Login flag="false" name="Login" number="1.3" optional="true" validity="Invalid"> <Check check="Label" result="login failed. Unknown user or wrong password(_)"/> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="testtest"/> <Button field="button_1.3"/> </Login> </Path> </Script>

5.2.1 | Metrics

To address the research questions above, we collected some metrics during the execution of the study. The first metric (**M2.1**) (related to question **RQ2.1**) keeps track of the amount of failures found. The second metric (**M2.2**) (related to question **RQ2.2**) saves information about the effort required to build test models or test scripts. The third metric (**M2.3**) (related to question **RQ2.2**) saves information about the effort required by PBGT during the configuration step.

5.2.2 | Subject systems

To answer the research questions posed, we selected a set of applications available on the Web, with source code available and with a wide set of features. We selected existing Web applications because we wanted to maintain independence in the performed study and also as a way of emphasize that PBGT approach can be applied to any Web application and not just some specific software that had been created by the authors for this purpose. Moreover, the selection was about applica-

TABLE 7 Case study findings concerning the modeling and configuration efforts and failures found

Subject system	Modeling and configuration efforts, minutes	Revealed failures
SS1.1	M: 30 C: 26	1
SS1.2	M: 44 C: 31	4

tions with source code available because we wanted to use fault seeding techniques to compare which of the approaches could find more failures. Thirdly, we searched for applications that contained numerous windows and widgets, and a reasonable set of features to have broader assessment of the capabilities of both testing tools. Therefore, we selected the following systems subject:

- **SS2.1** – Tudu Lists[†]. This is a simple application that manages todo lists. These lists can be accessed, edited, and shared. It is possible to create, edit, and delete lists and also todos. Further, todos can be filtered according to criteria, and Tudu also allows to manage user information. Moreover, these functionalities are only available upon successful authentication.
- **SS2.2** – Task Freak[‡]. This is a Web-based task manager. It allows to create, edit, and delete projects and also tasks. It is possible to filter them by several contexts (work, document, meeting, among others) and also to sort. It features an authentication mechanism (username and password). It also provides the ability to add comments to projects and tasks.
- **SS2.3** – iAddressBook[§]. This is a Web-based application to manage contacts. It allows to create, edit, and delete contacts; delete multiple contacts; create and delete categories for the contacts; add/remove contacts to/from categories; filter contacts; and search for contacts.

5.2.3 | Testing goals

The defined testing goals for all subject systems are listed in different tables, namely, Table 8 displays testing goals for SS2.1; Table 9 for SS2.2; and Table 10 displays the testing goals for SS2.3. These tables contain IDs that will be used to refer the testing goal in question and its description. These testing goals were provided to all the teams.

5.2.4 | Setup

This study was performed by 3 separate teams consisting of master students. The first team (to be referred to as Team 2.1) used the PBGT approach according to the following steps: construction of test models based on the defined testing goals, configuration of those models, selection/configuration of the TC generation algorithm, generation of TCs, and execution of TCs. The second team (to be referred to as Team 2.2) applied manual model-based TC generation to test the subject systems: designing of TCs based on the defined testing goals, recording the TCs with Selenium IDE tool, and executing TCs with Selenium IDE tool. Selenium IDE is a plugin that is installed in a Web browser and allows the tester to capture all his actions in a website and replicate them later on. The third team (to be referred to as Team 2.3) used a random testing tool to test the mutated versions of the applications. These teams have not been in contact with each other.

All testing approaches are black-box testing methods, as none of them has access to the application source code. They both perform the tests by following through a sequence of actions previously defined (a PARADIGM model in the case of PBGT and a test script in the case of Selenium).

TABLE 8 Defined testing goals for *Tudu Lists* (SS2.1)

ID	Description
TG2.1	Test the add new list functionality
TG2.2	Test the edit current list functionality
TG2.3	Test the delete current list functionality
TG2.4	Test the add quick Todo functionality
TG2.5	Test the add advanced Todo functionality
TG2.6	Test the edit Todo functionality
TG2.7	Test the delete Todo functionality
TG2.8	Test the complete Todo functionality
TG2.9	Test the show other Todos functionality
TG2.10	Test the delete completed Todos functionality
TG2.11	Test the "Next 4 days" filter
TG2.12	Test the "Assigned to me" filter
TG2.13	Test the authentication functionality
TG2.14	Test the edit user data functionality
TG2.15	Test the logout functionality

TABLE 9 Defined testing goals for *Task Freak* (SS2.2)

ID	Description
TG2.16	Test the login functionality
TG2.17	Test the logout functionality
TG2.18	Test the access user data page functionality
TG2.19	Test the delete user (as admin only) functionality
TG2.20	Test the see project functionality
TG2.21	Test create new project (as admin only) functionality
TG2.22	Test the edit project (as admin only) functionality
TG2.23	Test the delete project (as admin only) functionality
TG2.24	Test the add new task (by table and by menu) functionality
TG2.25	Test the edit task (by icon and by table entry) functionality
TG2.26	Test the delete task functionality
TG2.27	Test the change task completion status functionality
TG2.28	Test the add new comment functionality
TG2.29	Test the edit comment functionality
TG2.30	Test the sort tasks functionality
TG2.31	Test the filter by user functionality
TG2.32	Test the filter by context functionality

[†]http://www.julien&LWx02010;dubois.com/tudu&LWx02010;lists.html
[‡]http://www.taskfreak.com/
[§]http://iaddressbook.org/wiki/

TABLE 10 Defined testing goals for *iAddressBook* (SS2.3)

ID	Description
TG2.33	Test the select contact functionality
TG2.34	Test the add new contact functionality
TG2.35	Test the edit contact functionality
TG2.36	Test the delete contact functionality
TG2.37	Test the delete several contacts (bulk delete on contact list) functionality
TG2.38	Test the add to category functionality
TG2.39	Test the delete from category functionality
TG2.40	Test the create category functionality
TG2.41	Test the delete category functionality
TG2.42	Test the contact's initials filter functionality
TG2.43	Test the category filter functionality
TG2.44	Test the search functionality

This comparison featured all TC generation algorithms used by PBGT (*Default*, *Random*, and *Invalids*).

To answer the research question **RQ2.1**, we seeded faults in each Web application and measured how many failures each approach could find. In addition, we also used the TC generation techniques provided by PBGT (described in Section 4.3) to compare them.

To address **RQ2.2**, we measured the time, in minutes, that the teams required to create and configure the models (for the PBGT approach) and the time spent in creating scripts (for the manual approach).

To run the study, we provided the URL of the original version of each system and also the URL of the corresponding versions with the seeded faults to the teams. The original application was used to show the intended behavior. The procedure followed to perform this study is described below.

5.2.5 | Procedure

Firstly, we have prepared the environment for the execution of the study: we created altered versions of the original applications by means of a custom developed tool with the same behavior as *muJava* [56] using mutation operators. We restricted the set of mutant operators to arithmetic and logic operators [57]. We checked manually if the mutants corresponded to realistic failures. After that, each mutant was made available in a different URL. Then, we presented the applications to test to the teams in separate sessions. We gave the same set of testing goals to all teams and provided the URL of the original and altered versions of the 3 different applications.

The first team (PBGT) created models to fulfill the testing goals. Then, they configured the models based on each original version of the subject system, and then they executed the TCs in each mutated version of the subject systems.

The second team created a set of scripts, based on the testing goals, using Selenium IDE for each unaltered subject system. While navigating through the subject systems, Selenium IDE would automatically record each action they performed. After the scripts were recorded, they were replayed in each mutated version of the subject system.

The third team used a in-house random testing tool to test the mutated versions of the software application.

During the realization of the study, the teams gathered information about the time to build the model and the time to execute the TCs.

5.2.6 | Results

One example of a model written in PARADIGM for PBGT, regarding SS2.1, is illustrated in Figure 20. In Tudu Lists, the user is able to make a new user registration. To achieve the latter, he needs to access to the registration area. This is modeled with a Call UI Test Pattern (element number 1 labeled "Register Call"). Then, the user is redirected to a registration form (element number 2, labeled "Register") where he provides the necessary data to register a new user. To login, the user needs to select a link (called "Welcome") that will lead to the login screen. This is modeled with a Call UI Test Pattern (element number 10, labeled "MainPageCall"). The login screen is modeled as a Login UI Test Pattern (element number 3, labeled "Login"). After a successful login, the user is able to access, without any specific order (element number 4, labeled "Logged"), to his todos (element number 8, labeled "myTodos") or to access his user information (element number 6, labeled "myInfo"). The user is also able to perform logout (element number 9, labeled "Logout"). More details about the models, configurations, and generated TCs can be found in [55].

After building the model, the tester proceeds to its configuration. For each UI Test Pattern, the tester defines the input data and selects the checks to perform. A part of the configuration for the model displayed in Figure 20 is represented in Table 11.

FIGURE 20 Tudu Lists model in PARADIGM

5.2.7 | Findings

The teams had no major problems during the execution of the case study. They were able to fulfill all testing goals. The efforts required to build and configure models with PBGT and manual model based are displayed in Table 12 (**RQ2.2**).

Creating scripts (manual approach) took less time than crafting and configure models for PBGT. On average, from the universe of 3 subject systems, manual approach took 46 minutes less than PBGT. In detail, PBGT took extra 13 minutes for SS2.1 (Tudu Lists), 63 extra minutes for SS2.2 (TaskFreak), and 62 extra minutes for SS2.3 (iAddressBook).

The findings regarding the maximum of mutants killed for each subject system concerning PBGT, manual approach, and random testing are represented in Table 13 (**RQ1**). For SS2.1 (Tudu Lists), we seeded 35 faults; for SS2.2 (TaskFreak), 82 faults; and for SS2.3 (iAddressBook), 73 faults.

For instance, for SS2.1, the PBGT approach was able to kill a maximum of 32 mutants, the Manual approach was able to kill 16, and random testing killed only 1 mutant. Pattern-based GUI testing was able to kill additional 16 mutants when compared with Manual. Overall, PBGT was able to kill more mutants than Manual, making an average of 16 additional mutants killed. Regarding random testing, the difference is even higher because random testing was able to kill only an overall total of 14 mutants.

The results regarding the comparison of different PBGT test strategies (**RQ2.1**) can be seen in Table 14.

The table shows how many faults were seeded into the SUTs, how many mutants were killed, the average number of TCs required to kill 1 mutant and the average time (in minutes) to do so. For example, considering SS2.1, we seeded 35 faults, and the *Default* strategy was able to kill 24 mutants. In addition, the average number of TCs required to kill the latter mutants was 1.3, making an average of 9.1 minutes to achieve it.

5.2.8 | Threats to validity

For *external validity*, the threats are related with the number of subject systems used and the number of GUI testing approaches that are compared with PBGT. The number of subject systems is limited, but are representative of modern open-source Web applications, with different functionalities, and were implemented in different programming languages. In addition, the results gathered were obtained by applying TCs over 190 mutants, which requires analysis time. For *internal valid-*

TABLE 11 Configuration details for the model from Figure 20

Name	Type	Validity	Variables	Check
RegisterCall	Call	NA	{}	{Change to Page "New user registration"}
MainPageCall	Call	NA	{}	{Change to Page "Welcome to Tudu Lists"}
Login	Login	Valid	{[username, "john doe"]; [password, "pass"]}	{}
		Invalid	{[username, "test"]; [password, "test"]}	{Label "Welcome to Tudu Lists!"}
myInfoCall	Call	NA	{}	{Change to Page "Manage user information"}
myTodosCall	Call	NA	{}	{Change to Page "Actions"}
Logout	Call	NA	{}	{Change to Page "You have left Tudu Lists."}

TABLE 12 Efforts required to build and configure models/scripts for each subject system

Subject system	Testing approach	Minutes	Difference (PBGT—Manual)
Tudu Lists	PBGT	73	+13 minutes
	Manual	60	
TaskFreak	PBGT	164	+63 minutes
	Manual	101	
iAddressBook	PBGT	137	+62 minutes
	Manual	75	

TABLE 13 Maximum number of mutants killed for each subject system

Subject system	Testing approach	Seeded faults	Mutants killed	PBGT—Manual
Tudu Lists	PBGT	35	32	+16
	Manual		16	
	Random		1	
TaskFreak	PBGT	82	73	+20
	Manual		53	
	Random		9	
iAddressBook	PBGT	73	68	+12
	Manual		56	
	Random		4	

Abbreviation: PBGT, pattern-based GUI testing.

TABLE 14 Number of mutants killed in each PBGT test strategy

Subject system	Strategy	Faults	Mutants killed	TC/mut. (avg)	Time/mut. (avg)
Tudu Lists	Default	35	24	1.3	9.1
	Random		32	5.2	26.5
	Invalids		27	3.7	18.5
TaskFreak	Default	82	59	2.3	18.1
	Random		73	6.1	51.2
	Invalids		69	4.0	33.8
iAddressBook	Default	73	60	1.2	8.8
	Random		68	5.7	41.2
	Invalids		67	3.4	25.7

Abbreviation: PBGT, pattern-based GUI testing; TC, test case.

ity, the threat can be related to the measurement of modeling effort. Nevertheless, the work was done by different teams unfamiliar with testing approaches and tools and therefore were teams who started the work from scratch. If the study is repeated with teams with some prior knowledge of the different testing approach, the time spent in modeling may be different, but in that situation, it will be possible to argue that the prior knowledge may be different and the results may not be valuable either.

6 | DISCUSSION

At the end of the case studies, it was possible to analyze the results to answer the research questions. After case study 1, it became apparent that testers without any knowledge of PBGT approach could begin to use and obtain the benefits from its use immediately after a training session of 2 hours. This also indicates that the approach is easy to learn. Furthermore, during the experiments, the testers were able to successfully accomplish the testing goals and were able, by themselves, to create and configure models. In addition, the testers were able to find failures in fielded applications.

At the end of case study 2, it was possible to notice that the effort to build PARADIGM test models is greater than to create test scripts in Selenium. The extra effort needed to model the application in PBGT compensates in the number of mutants killed. When comparing PBGT with random testing, it is possible to see that random testing is able to kill a small set of mutants because it only kills the ones that crash the application under test.

Some mutants killed by PBGT were not killed by Selenium IDE. For instance, in the TaskFreak application (SS 2.2), there is a specific mutant that allows a user to create tasks for a project that does not belong to that specific user (Figure 21).

Typically, the tester first selects the user, and then he selects the project. If that specific user does not belong to the project, the user will be automatically changed to a default one. This mutant prevents the change. Since our Selenium scripts follows a linear approach, ie, fills in the fields in a regular order, this mutant is not killed. However, since these fields are marked as "AnyOrder" in the PARADIGM model, there are several TCs in which the user field is altered first than the project field. Furthermore, some mutants affected Web elements that Selenium IDE could not find in the Web page. For instance, in TaskFreak, when the user changes the status (percentage of completion) of a task, color images appear (as illustrated in Figure 22, featuring the colored squares on the right). In this figure, the "teste" task is at 40% (each square located at the most right of Figure 22 corresponds to 20%). The mutants that change the behavior of the color images cannot be killed by Selenium.

Although PBGT uses Selenium libraries to identify Web elements, it can also use Sikuli to identify and interact with Web elements through image recognition, so it is able to kill the mutant. Sikuli takes a picture of the element during the mapping to find the element during the test execution. This is particularly useful for applications that work within their own black-box in the browser, such as Adobe Flash [58] or Microsoft Silverlight [59] applications. Also, if a pop-up window is generated, Selenium IDE is unable to interact with it, whether PBGT will interact using Sikuli.

Besides mutants killed by PBGT but not killed by Selenium, there are also some (1) mutants that were not killed by any of the 2 approaches and (2) few mutants that were killed only by Selenium. The former ones (1) are generically related to functionality not tested by the TCs and changes reflected by Javascript. The latter ones (2) were generically related to data maintained between sessions. Pattern-based GUI testing does not check this because it opens the Web browser in the beginning of a TC and closes it at the end of a TC.

Despite being able to interact with other controls and, therefore, being able to further test the behavior of the application, the use of Sikuli is not the only advantage of PBGT nor the only reason to kill more mutants. An important issue is undoubtedly the set of TCs generated by PBGT as opposed to the number of scripts built with Selenium. While the Selenium team created an average of 5 to 7 scripts to fulfill the test goals defined for each Web application, the PBGT tool could generate hundreds of different TCs from each model (for Tudu Lists alone, 245 TCs were generated). This allowed to test sequences of actions that were not tested by Selenium scripts.

Another important aspect is the set of checks that PBGT models and Selenium scripts had. The PBGT team after the construction of the PARADIGM models followed through the configuration phase defining, among others, the checks to run in each UI Test Pattern within the model. With Selenium, the checks are defined along the construction of the test script. We noticed that PBGT models (and consequently the TCs generated from them) had more checks than the Selenium scripts. It seems to us that having an independent phase after building the model contributes to have a wider view of the testing goals and think more thoroughly about the checks to run in each point of the model.

With PBGT, test data are also defined during the configuration phase. After constructing a PARADIGM model, it is easy to continuously add different input data and checks, while the model stays the same. By adding more configurations, it may be possible to increase the number of tests cases and augment the percentage of code coverage. Within Selenium, if a tester wants to add a different set of inputs and checks, he has to create different test scripts, or alternatively, he can convert scripts to a programming language (eg, Java) and then change the code to build parameterized scripts that can be executed with different input data. These parameterized scripts are difficult to maintain when the original scripts require modifications.

We had not performed code coverage analysis in the experiments because PBGT aims to test only common behavior (UI patterns), so the percentage of code coverage would probably be low. In addition, at this moment, PBGT can only perform code coverage analysis over PHP software applications, which is not the case of the applications used in the case studies. However, this type of analysis can be interesting as how to move forward with additional testing through PBGT complementary testing techniques/approaches. The type of coverage analysis supported by PBGT at this moment can be seen in [60].

Concerning the comparison of different strategies for TC generation, and more specifically the comparison of *Invalids* with *Default* strategies, we could verify that *Invalids* strategy killed most of the mutants that were killed by *Default* strategy. However, it required more TCs to achieve it. Nevertheless, the *Invalids* could kill some mutants that *Default* could not kill. These were mutants that affected the authentication functionality, allowing invalid users to access private data, and incorrect data input verification.

Comparing the *Default* algorithm with the *Random* one, it was possible to see that *Random* could kill more mutants than the *Default* strategy (it killed some that were only killed by the *Invalids* strategy), but it would take more time and TCs to kill them. Also, since it is totally random, it repeated some TCs, and thus spending more time to kill different mutants.

In conclusion, the *Default* strategy can kill a significant number of mutants in a short time, while the *Random* strategy kills more mutants, but it takes more time. Also, since the *Default* strategy only runs 1 TC per test path, it always picks the first set of configurations the user defined. Most of the time the first set of configurations defined is a valid set, to ensure the system works as expected. Additional configurations usually test the application with more unexpected data, such as invalid characters, different data types (eg, input a string where a number was expected), and invalid credentials (such as username and password). These are the type of data that is more error prone. With *Random* strategy, the algorithm can choose these types of configurations as well as the valid ones, increasing the probability of killing a mutant. The *Invalids* should be used when the tester wants to check authentication or validation issues in the Web application.

The failures found by PBGT can be classified in 2 main groups: (1) the behavior is different than the expected and (2) the test driver tries to interact with GUI objects (widgets) that do not exist or are disabled. In the first group, there are the failures caused by widgets, such as buttons, text boxes, and links, that stop working or function in a different way than expected; failures preventing the entry of correct data; authentication failures that allow access to restricted data; and data inputs ignored. In the second group, we can find failures caused by dynamic content, which ceased to exist, and the blocking of the application caused by, for example, loss of connectivity to the database.

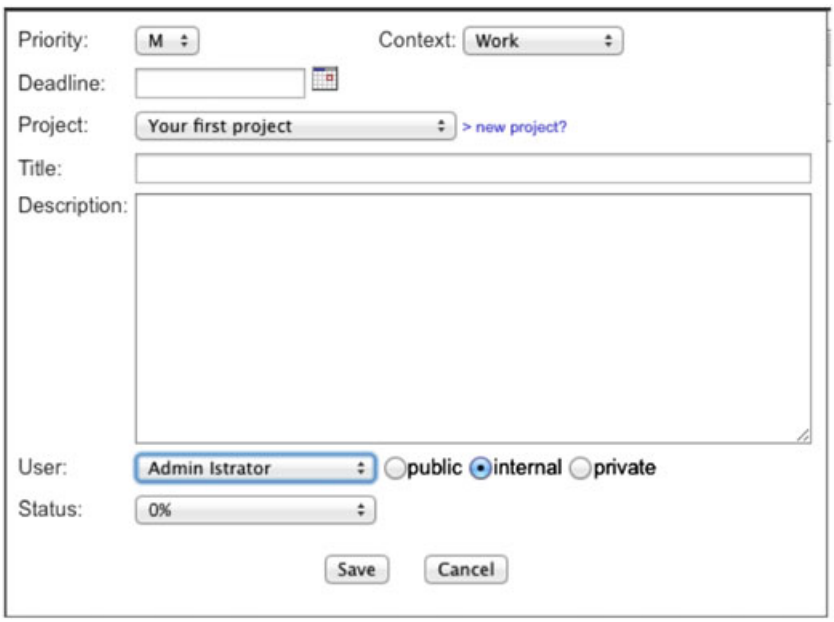

FIGURE 21 TaskFreak graphical user interface featuring project properties

FIGURE 22 TaskFreak graphical user interface featuring the details of a project

Besides the experiments described before, the PBGT modeling language (PARADIGM) was compared with other modeling languages (Spec# and VAN4GUIM) [8] in an industrial context. Professional testers were asked to model 5 different subject systems, and the time required to model and configure the models in the different approaches were registered. The modeling effort was lower with PARADIGM, but regarding the configuration effort, the difference was not so clear. However, the feedback obtained through a questionnaire was promising.

Finally, we did not compare PBGT with any other automatic model-based test generation approach, such as the one used by GUITAR. A fundamental reason for skipping this comparison is that the approaches are incomparable. Pattern-based GUI testing helps us to generate test input (sequences of events) as well as the test oracle (the mechanism that determines whether a TC passed or failed). The GUITAR, on the other hand, generates only the test input; for the test oracle, it relies on software crashes, which in our case studies were irrelevant, or reference testing, in which it compares the full current GUI state with that of a reference "golden" version available during regression testing, again irrelevant for our studies. Our long-term goal is to incorporate the PBGT approach with the algorithms used by GUITAR, so as to enhance both approaches.

7 | CONCLUSIONS

In this paper, we presented a new MBGT technique, entitled PBGT, which aims to diminish the testing effort and promote reuse of GUI testing strategies. Further, we developed the notion of UI Test Patterns, which define generic test strategies that are able to test different implementations of UI Patterns. User interface test patterns represent behavioral elements within PARADIGM—a DSL created specifically for PBGT—that allows to craft models describing testing goals. Models are built and configured within PARADIGM Modeling Environment. Test cases are generated automatically from the models. In addition, we provide a detailed description of the UI Test Patterns according to best practices and guidelines from the field of Pattern Languages.

We empirically evaluated the PBGT approach by conducting 2 case studies according to Yin's guidelines. In the first study, we assessed the PBGT failure detection capabilities in 2 fielded systems. Results indicate the effectiveness of the PBGT approach, since testers were able to find failures in those systems and within short time. In the second study, the goal was to compare the PBGT approach with random testing and with the Manual model-based generation of TCs to assess their ability in finding failures; measure the time required to start executing tests; and evaluate the PBGT's TC generation strategies (*Default*, *Random*, and *Invalids*), in failures found and time required to execute. Thus, we seeded faults in 3 Web-based applications. Results indicate that despite the fact that PBGT spent more time in configure the models (when compared with Manual approach), it was capable of finding more failures. It was also possible to conclude that PBGT is able to kill much more mutants that random testing because the latter is only able to kill mutants that crash the application under test. Regarding TC generation, the *Random* strategy was able to find more failures, but it consumed more time. This result could be different if we had configured the *Default* strategy so to execute more than 1 TC per test path.

One significant aspect of PBGT is the moment in which testers can start their activities. Although the case studies were based on existing Web applications, the PBGT approach can also be useful during development of Web applications starting the construction of the PARADIGM models when eliciting requirements. The PARADIGM models can be built during the requirements elicitation phase when there is no code yet. Along the development, the PARADIGM models can be updated (if needed) and configured. The developed GUI is only required during the mapping phase.

Regarding future work, we intend to explore more about reverse engineering approached to extract automatically UI Test Patterns (and possible some configurations of those patterns and mapping) from Web applications to diminish further the modeling effort. This is a test scenario that can be useful when the application under test already exists. Additionally, we intend to add more UI Test Patterns to the PARADIGM language, to keep up with latest progress in the field of GUI trends, and include a constraint language to define, in a more flexible way, the checks to perform during the configuration step of the UITPs.

We also aim to define different levels of testing goals so as to allow the tester to define the level of testing strategies he wants to apply. For example, in the case of the login pattern, it will be possible to choose only a strategy to test valid and invalid authentication data or to test additionally different types of invalid cases (eg, blank username, blank password, etc). This way, defining only 1 configuration for valid and another for invalid choosing a strong testing goal approach would result in a not complete coverage of the testing goals of the model. This information will be useful for the tester to help him configure the tests according to the test strategy chosen.

One issue that can be improved in the PARADIGM-ME is its usability to facilitate even more the configuration of UI Test Patterns. The generated reports (failures found) will also be subject of enhancement. Another functionality on the product roadmap is the ability to share UI Test Patterns among PARADIGM-ME users (testers), allowing to promote collaboration among testers. At this moment, PARADIGM-ME supports testing of Web and mobile [61] applications. Some videos of the tool can be seen on [55]. In the future, we aim to be able to test desktop application as well. Yet, concerning the mobile domain, we intend to adapt and evolve the set of UI Test Patterns to cover the particularities of mobile applications, for example, other forms of interaction such as long press.

Test case generation in PBGT is an area that can be improved. We aim to investigate and analyze further test case generation algorithms, for instance [51, 52], including additional test strategies and filtering options and identify where each of them can be more effective.

Finally, we intend to promote the PBGT approach in the testing community and direct the approach towards industry adoption. The idea is to promote sessions at software companies, so test teams could see the value of PBGT, its benefits and adopt it.

ACKNOWLEDGEMENTS

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. N. Nyman, *Using Monkey Test Tools*, Software Test. Qual. Eng. Mag. **2000-01** (2000), 18–21.
2. 2015. http://www.seleniumhq.org/projects/ide/. Accessed February 2015.
3. S. Nedyalkova and J. Bernardino, Open source capture and replay tools comparison, *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, ACM, New York, NY, USA, 2013, pp. 117–119.
4. BN Nguyen, B. Robbins, I. Banerjee, and A. M. Memon, *Guitar: an innovative tool for automated testing of GUI-driven software*, Autom. Softw. Eng. **21** (2014), no. 1, 65–105.
5. H Reza, S. Endapally, and E. Grant, in A model-based approach for testing gui using hierarchical predicate transition nets, Information Technology, 2007. ITNG '07. Fourth International Conference on, Las Vegas, 2007, pp. 366–370.
6. R Moreira, A. Paiva, and A. Memon, in A pattern-based approach for gui modeling and testing, Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, Pasadena, CA, USA, 2013, pp. 288–297.
7. *Model-based testing*, 2015. http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html. Accessed December 2015.
8. R. M. L. M. Moreira and A. C. R. Paiva, A GUI Modeling DSL for Pattern-Based GUI Testing - PARADIGM, *9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, LA Maciaszek and J Filipe, (eds.), SciTePress, Lisbon, Portugal, 2014, pp. 1–10.
9. R. M. L. M. Moreira and A. C. R. Paiva, Towards a pattern language for model-based gui testing, *Proceedings of the 19th European Conference on Pattern Languages of Programs*, EuroPLoP '14, ACM, New York, NY, USA, 2014, pp. 26:1–26:8.
10. M. Nabuco and A. C. R. Paiva, Model-based test case generation for web applications, *Computational Science and its Applications ICCSA 2014*, B Murgante, S Misra, A. Rocha, C Torre, J Rocha, M Falco, D Taniar, B Apduhan, and O Gervasi, (eds.), Lecture Notes in Computer Science, Vol. **8584**, Springer International Publishing, Guimaraes, Portugal, 2014, pp. 248–262. doi:10.1007/978-3-319-09153-2_19.
11. S. Arlt, C. Bertolini, S. Pahl, and M. Schäf, *Trends in Model-based GUI Testing*, Adv. Comput. **86** (2012), 183–222.
12. K. Li and M. Wu, *Effective GUI Testing Automation: Developing an Automated GUI Testing Tool*, Wiley, Hoboken, NJ, 2006.
13. M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj, in A survey of model-driven testing techniques, Quality Software, 2009. QSIC '09. 9th International Conference on, Jeju 2009, pp. 167–172.
14. M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
15. A. M. Memon, M. L. Soffa, and M. E. Pollack, Coverage Criteria for GUI Testing, *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, Vienna, Austria, 2001, pp. 256–267.
16. A. Kervinen, M. Maunumaa, T. Paa¨kkönen, and M. Katara, Model-based testing through a GUI, *In Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), number 3997 in Lecture Notes in Computer Science*, 2006, pp. 16–31.
17. R. Shehady and D. Siewiorek, in A method to automate user interface testing using variable finite state machines, Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on, Seattle, WA, USA, 1997, pp. 80–88.
18. Y. Miao and X. Yang, in An fsm based gui test automation model, Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on, Singapore 2010, pp. 120–126.
19. H. Reza, K. Ogaard, and A. Malge, A model Based Testing Technique to Test Web Applications Using Statecharts, *Proceedings of the Fifth International Conference on Information Technology: New Generations*, ITNG '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 183–188.
20. Microsoft, *Spec# - Microsoft Research*, 2013. http://research.microsoft.com/en-us/projects/specsharp/ [Accessed January, 2013].
21. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L Nachmanson, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, *Formal methods and testing*, RM Hierons, JP Bowen, and M Harman, (eds.), Lecture Notes in Computer Science, Springer, Berlin, April 21, 2008, pp. 39–76.
22. A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, in A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing, Manchester, UK, 2005, pp. 450–464.
23. R. M. L. M. Moreira and A. C. R. Paiva, in Visual abstract notation for gui modelling and testing - VAN4GUIM, ICSOFT 2008 - Proceedings of the Third International Conference on Software and data Technologies, Volume SE/MUSE/GSDCA, Porto, Portugal, August July 5, pp. 104–111.
24. E. Elsaka, W. Moustafa, B. Nguyen, and A. Memon, in Using methods and measures from network analysis for gui testing, Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, Paris, 2010, pp. 240–246.
25. *Welie.com - patterns in interaction design*, 2016. http://www.welie.com/patterns/index.php. Accessed March 2016.
26. M. Cunha, A. C. R. Paiva, H. Ferreira, and R. Abreu, Pettool: A pattern-based gui testing tool, *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, Vol. **1**, 2010, pp. V1–202–V1–206.
27. Apple, *Cocoa bindings programming topics: Creating a master-detail interface*, 2014. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaBindings/Tasks/masterdetail.html. Accessed April 2014.
28. R. M. L. M. Moreira and A. C. R. Paiva, in A novel approach using alloy in domain-specific language engineering, Proceedings of the 3th International Conference on Model-Driven Engineering and Software Development - Modelsward, Angers, Loire Valley, France, 2015, pp. 157–164.
29. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, 2nd Revised edition, MIT Press, London, England, 2011.
30. R. M. L. M. Moreira and A. C. R. Paiva, Pbgt tool: An integrated modeling and testing environment for pattern-based gui testing, *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, ACM, New York, NY, USA, 2014, pp. 863–866.
31. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed., Addison-Wesley Professional, Salt Lake City, Utah, USA, 2009.
32. *Selenium webdriver*, 2014. http://docs.seleniumhq.org/projects/webdriver/. Accessed March 2014.
33. SikuliLab-Department of Computer Science - University of Colorado Boulder, *Sikuli script - home*, 2014. http://www.sikuli.org/ [Accessed April, 2014].
34. M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*, John Wiley & Sons, New York, USA, 2005.
35. J. Tidwell, *Designing Interfaces*, O'Reilly, Sebastopol, CA, 2011.
36. Yahoo!, *Yahoo! Design Pattern Library*, 2012. http://developer.yahoo.com/ypatterns [Accessed December, 2013].
37. Patternry, *Patternry Open - A Free Front-End Resource — Patternry*, 2009. http://patternry.com/patterns/ [Accessed January, 2014].
38. A. Toxboe, *Design patterns*, 2013. http://ui-patterns.com/patterns/ [Accessed January, 2014].

39. M. van Welie, *Interaction Design Pattern Library*, 2008. http://www.welie.com/patterns [Accessed January, 2014].
40. UASP IDT, *Pattern Browser*, 2008. http://patternbrowser.org/code/pattern/pattern.php [Accessed January, 2014].
41. C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, Oxford, 1977.
42. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design patterns: Elements of reusable object-oriented software*, 1st ed., Addison-Wesley Professional, Salt Lake City, Utah, USA, 1994.
43. R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 2003.
44. G. Meszaros and J. Doble, in A Pattern Language for Pattern Writing, The 3rd Pattern Languages of Programming Conference, Monticello, Illinois, 1996, pp. 1–33.
45. Stattrek, *T distribution calculator*, 2014. http://stattrek.com/online-calculator/t-distribution.aspx [Accessed March, 2014].
46. R. Raszka, *Pttrns - Mobile User Interface Patterns*, 2011. http://pttrns.com/ [Accessed January, 2014].
47. Facebook, *Welcome to Facebook - Log In, Sign Up or Learn More*, 2014. http://www.facebook.com [Accessed January, 2014].
48. mobile.de, *mobile.de - Germany's Biggest Vehicle Marketplace Online. Search, Buy and Sell Used and New Vehicles*, 2014. http://www.mobile.de [Accessed November, 2013].
49. GPUpdate, *Formula 1, motogp, gp2, gp3, f3 and indycar news, photos and much more at gpupdate.net*, 2014. http://www.gpupdate.net/en/search/ [Accessed March, 2014].
50. Telerik, *Telerik Mobile App Development Platform, .NET UI Controls, Web, Mobile, Desktop Development Tools*, 2014. www.telerik.com [Accessed January, 2014].
51. T. J. Ostrand and M. J. Balcer, *The category-partition method for specifying and generating fuctional tests*, Commun. ACM **31** (1988), no. 6, 676–686.
52. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, *An orchestrated survey of methodologies for automated software test case generation*, J. Syst. Softw. **86** (2013), no. 8, 1978–2001.
53. R. K. Yin, *Case Study Research: Design and Methods*, Vol. **5**, Sage, Thousand Oaks, CA, USA, 2009.
54. S. Hung, *Australian charts portal*. http://australian-charts.com [Accessed November, 2013].
55. *PBGT - Pattern Based GUI Testing wiki*, 2015. http://paginas.fe.up.pt/~apaiva/pbgtwiki/doku.php [Accessed March, 2015].
56. Y.-S. Ma, J. Offutt, and Y. R. Kwon, *MuJava: an automated class mutation system*, Software Test. Verification Reliab. **15** (2005), no. 2, 97–133.
57. L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, *Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation*, IEEE Trans. Softw. Eng. **40** (2014), no. 1, 23–42.
58. Adobe, *Flash Player | Adobe Flash Player | Overview*, 2014. http://www.adobe.com/products/flashplayer.html [Accessed January, 2014].
59. Microsoft, *Microsoft Silverlight*, 2014. http://www.microsoft.com/silverlight/ [Accessed February, 2014].
60. L. Vilela and A. Paiva, in Paradigm-cov: A multimensional test coverage analysis tool, Information Systems and Technologies (CISTI), 2014 9th Iberian Conference on, Barcelona, 2014, pp. 1–7.
61. P. Costa, A. Paiva, and M. Nabuco, in Pattern based gui testing for mobile applications, Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the, Guimaraes, 2014, pp. 66–74.

How to cite this article: Moreira RMLM, Paiva AC, Nabuco M, Memon A. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Softw Test Verif Reliab*. 2017;**27**:e1629. https://doi.org/10.1002/stvr.1629.