PETTool: A Pattern-Based GUI Testing Tool

Marco Cunha[†], **Ana C. R. Paiva**[†], **Hugo Sereno Ferreira**[†‡] and **Rui Abreu**[†]

[†]Department of Informatics Engineering
Faculty of Engineering, University of Porto, Portugal

[‡]INESC Porto
Porto, Portugal

e-mail: {ei05048,apaiva,hugosf}@fe.up.pt , rui@computer.org

Abstract— **Nowadays, the usage of graphical user interfaces (GUIs) in order to ease the interaction with software applications is preferred over command line interfaces. Despite recent advances in software testing, GUIs are still tested in a complete ad-hoc, manual fashion, with little support from (industrial) testing tools. Automating the process of testing GUIs has additional challenges when compared to command-line applications. This paper presents an approach for GUI (semi-automated) testing which uses knowledge of the common behaviour of a GUI. To do so, the most common aspects in a GUI are identified and then a suite of test cases is automatically generated and executed. To validate our approach, we have run it against well known web-based applications, such as GMail.**

Keywords - Software testing, Graphical User Interfaces, patterns.

I. Introduction

Graphical User Interface (GUI) software testing is the process of testing a software application with a graphical front-end to guarantee that it meets its specification. Given the myriad of possible (human-computer) interactions in such a user interface, GUI software testing is a rather expensive, cumbersome phase of the whole software development cycle. For instance, a form containing *n* interface controls requires a factorial number (*n!*) of test cases to test all possible combinations [11].

In GUI software testing, a good test suite must not only cover all functionality of the system but also ensure that the GUI itself is fully exercised (i.e., tested thoroughly). Creating such a test suite is overly difficult because the software tester has to deal with the (i) domain size as well as (ii) the action sequences. To illustrate the importance of these points for GUI testing, as an example, consider the fact that even a very small program, such as Microsoft WordPad[1], has 325 possible GUI operations [6]. Action sequencing is also a problem because some functionality of the system requires a deterministic sequence of GUI events.

Given these GUI's idiosyncrasies, manually generating a comprehensive suite of test cases is extremely difficult (not to say insurmountable). Hence, these issues have driven the GUI testing problem domain towards (semi/full) automation.

User-computer interaction – viz. graphical user interfaces (GUIs) – represents the primary conduit through which a user "perceives" the system. This area has been actively studied by giving particular focus on what is defined as "usability", and several pattern languages exist to describe the reusable elements of user-computer interaction [10]. Most of today GUIs are built around these patterns, and thus their behaviour is expected to vary only slightly.

It is this specific property of GUIs that pushed us to extend the notion of testing to a "pattern-based testing". As said above, approaching the problem of testing by aiming for completeness would require either a formal model of the underlying application, or exhaustive analysis. Other approaches, such as unit-testing, often require access to the source-code (or at least to a public API). But if we take into account that most GUI behaviour is recurrent, we are able to devise recurrent schemes to test that behaviour. These elements would constitute a catalog of patterns for GUI testing, which although not aiming for completeness, it specifically targets common, recurrent behaviour of software.

In this paper, we present a pattern-based approach for (semi-)automating GUI testing, dubbed PETTOOL. The approach identifies patterns of GUI behaviour and, for each one, provides generic GUI testing solutions based on known software testing techniques. These generic solutions establish a map between the inputs of the testing tool and the GUI controls where the tests will then be executed. Finally, after running the real GUI against a set of automatically generated test suite, the tool generates a report with the observed results (pass/fail information, error/successful messages are included in such report). We have tested our approach against well-known webmail applications, such as GMail, and a real estate agency website.

This paper is organized as follows. We start off by presenting a motivational example for pattern-based GUI testing in Section II. In Section III we motivate how test patterns can be used to automate GUI testing. In Section IV we propose a solution towards automating GUI testing, and we describe some case studies in Section V. Related Work is given in Section VI. In Section VII we conclude and discuss future work.

II. Motivational Example

One common artifact in both applications and websites is the authentication form. The purpose of the authentication form is to serve as a user access controller, and is typically composed by two text boxes (*username* and *password*) and one button to validate the data entered.

Despite being a very common practice, it has in fact many possible implementations and outcomes. For example,

[1] WordPad is a basic word processor shipped with almost all versions of Microsoft Windows. It is much simpler than Microsoft Word.

given an authentication failure, the interface could give no feedback and remain in the form (perhaps the worst scenario with respect to *usability* principles). Sometimes, it shows a message inside the page (e.g., "*Please enter your username*") but these messages could be shown in a different window and not in the main form. Another possibility consists in disabling the submit button when the fields are not filled, amongst others.

In GMail[2], whenever an error occurs, the message is shown inside the form, below the field that has caused the error. Nonetheless, other authentication interfaces might have different behaviours. The authentication form in the Webmail[3] front-end used at the Faculty of Engineering of University of Porto (FEUP)[4] for instance, has a different behaviour: error messages about missing information are shown outside the form, but messages about invalid data are shown inside it.

These are examples of how a simple and common mechanism such as the authentication form can have so many differences between implementations. Despite the feature under test being basically the same, and most of the expected behaviour being just variants (and compositions) of common outcomes, these differences lead to difficulties in reusing test cases, hence reducing the efficiency of the testing process.

Our approach tries to use the knowledge of the most common patterns in GUIs in order to create generic tests for them (which, incidentally, reveal themselves as test patterns).

In the next sections, it will be shown how this approach can be used to test some recurrent GUI behaviour such as the authentication form.

III. GUI AND TEST PATTERNS

GUIs are typically composed by several GUI patterns. A detailed catalog of patterns can be found in [8], [10]. In this section, we will pick a subset of these patterns as depicted in Fig. 1, and explain what strategies can be used to test them.

- FORM. A page with Data Entry Fields is supposed to be filled by the user, and it is a fundamental user-interface element of most applications and websites. A form by itself does not have any generic testing strategy, but it denotes a *composition*. Therefore, in order to test a form it is necessary to identify its components and their corresponding behaviour.

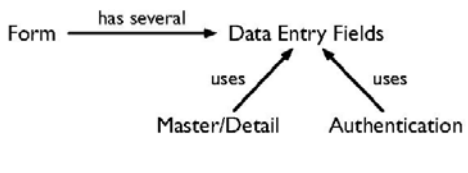

Figure 1 Relation between patterns. A FORM is usually composed of many DATA ENTRY FIELDS. Those fields are used to define higher level patterns such as the MASTER/DETAIL and AUTHENTICATION patterns.

2 http://www.gmail.com

3 http://webmail.fe.up.pt

4 http://www.fe.up.pt

  Each component can then be tested both individually (e.g., to test if an invalid input is detected properly) and in combination with other components (e.g., to evaluate if some errors can be undetected when combined).
- DATA ENTRY FIELD. Fields present are the primary means for a user to enter information. Data Entry Fields can be defined as mandatory according to the relevance of the information they collect. Therefore, to test the behaviour of a Data Entry Field, equivalence class partitioning with boundary value analysis [2] can be used as a testing strategy, where inputs are divided in classes. Elements from each class must obey to some principles: (i) values from the same class must lead to the same behaviour, (ii) an error found with one element of the class could be found by any other element of the class, (iii) testing the application with one element of each class is enough to be confident that it works properly. After the class partitioning, the tester should use valid, invalid, and boundary values for each class.
- AUTHENTICATION. An authentication mechanism aims to allow users to prove they are who they say. It is typically used in situations where access to information is restricted or where the information to be shown is different among users. Commonly, this pattern is implemented by a GUI with two text-boxes and a submit button. Therefore, the equivalence class partitioning can also be used to test an Authentication pattern. In this case, three classes can be identified which leads to three tests: (i) the correct username/password is entered, (ii) a wrong username is entered and (iii) a valid username is entered but the password is wrong.
- MASTER/DETAIL. Sometimes, there are pairs of fields which are related, where one of these fields changes its value range w.r.t another value chosen in the other field. The latter is called the *Master*, and the former *Detail*. Known uses of this pattern include registration forms where the user must select a continent and then a country located in that continent. Every time the continent chosen changes, the set of countries is updated accordingly. Once again, equivalence class partitioning can be applied as a testing strategy in this pattern. In order to check if the pattern is well implemented, there are three tests that can be made for a specific Master value: (i) the Detail contains a specific value, (ii) the Detail contains only a specific set of values and (iii) the Detail does not contain a value.

The aforementioned strategies suggest that, for a particular user-interface, the number of tests generated is not so related to the number of patterns, but more to the tests defined in each pattern. For instance, if we only have one Data Entry Field with two valid values and three invalid values, five tests will be executed despite only one test strategy was used.

IV. IV. PETTOOL

Currently, PETTOOL[5] is capable to generate and execute tests for some patterns such as the FORM, the DATA ENTRY FIELD the AUTHENTICATION and the MASTER/DETAIL. The current architectural structure of the tool is the one presented in the Fig. 2.

Controls inside GUIs (e.g., *textboxes*) may be used by patterns. For instance, a Data Entry Field needs a control which allows user to enter the needed information. GUIs also have forms that contain behaviour patterns. For example, in a registration form, there are always Data Entry Fields.

Associated with each behavioural pattern there is a set of tests that must be executed to exercise its functionality. Each test is related to valid or invalid inputs, so different tests might expect different behaviours. In order to allow different outcomes for different tests, there is a set of Expected Behaviours associated with each behavioural pattern. Those behaviours define what is expected to happen in a certain situation (e.g., when testing the Authentication pattern with wrong inputs, it is possible to specify that the expected behaviour is to remain in the same form).

In order to present how PETTool combines different patterns to generate the tests, the authentication form will be used as an example. This example contains the form, two instances of the Data Entry Field and the Authentication patterns. According to the test strategy defined in the previous section, the tests that must be executed are present in Table I.

The first four tests are related to the Data Entry Field patterns. In this case, since both username and password are mandatory, it is only necessary to test two cases in each data entry field: valid information entered and field left blank. In this scenario, invalid information does not need to be tested since the authentication pattern is the one responsible to check user information. The last three tests are related to the authentication pattern. It is possible to see that test 4 and test 7 are redundant. In this case, PETTool identifies both data entry fields as components of the authentication pattern, skips the test 4 and maintains test 7. Next section will show how PETTool can be used to test two common GUI patterns, namely AUTHENTICATION and MASTER/DETAIL, and how test results are presented.

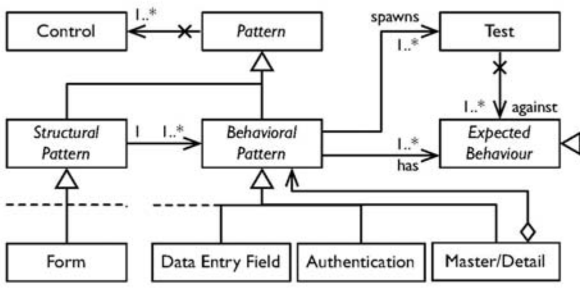

Figure 2 Architectural Structure of PETTool

TABLE I Tests for an authentication form

	Tests generated						
	Data Entry Field Patterns				***Authentication Patterns***		
	1	***2***	***3***	***4***	***5***	***6***	***7***
Login Field	Valid Value	-	-	Valid Value	Valid Value	Invalid Value	Valid Value
Password Field	-	Valid Value	-	Valid Value	Invalid Value	Valid Value	Valid Value

5 Acronym for *PattErn Testing Tool*.

V. Case Studies

This section presents how PETTool can be used to test the authentication form of GMail and a MASTER/DETAIL pattern in the website of a real estate agency called ERA[6].

A. GMail

In order to test the authentication form of GMail, the tester has to point out the form which is going to be tested (click on the ”Select” button labeled 1.1 in Fig. 3 and then click on the GMail form labeled 1.2) and point out the submit button (click on the ”Select” button labeled 2.1 and then click on the ”Sign in” button labeled 2.2).

Afterwards, the tester needs to identify the behaviour patterns within such form. GMail’s authentication form has two mandatory Data Entry Fields: Username and Password. When the Username field is left blank, the GMail shows a message: “Enter your email address.”. The same happens with the Password field, however, the message shown is different: “Enter your password.”. The behaviour expected when the authentication succeeds and when it does not succeed (e.g., because one of the fields is left blank or because the input values do not correspond to a valid user) can be described using the PETTool (Fig. 4). In GMail, when wrong data is entered, the text “The username or password you entered is incorrect.” is shown inside the form, otherwise the user is redirected to another page. The tester has to provide both a valid and invalid login data. This information is used to test the three possible scenarios of this pattern: (i) valid user information, (ii) valid login with wrong password and (iii) invalid login. Associated with each of the previous patterns there is a Response Time. This is a way to indicate how long the tool must wait before checking the test results. For instance, GUIs usually take longer to check invalid user information than to respond to an empty login. This way, the tests can be (i) faster and (ii) be used to check the response time of the GUI under certain circumstances. After performing these configuration steps, the tool generates test cases

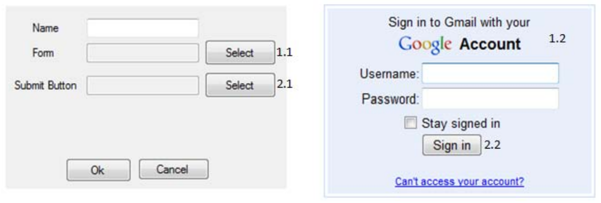

Figure 3 Instantiation of the FORM Pattern in GMail.

6 http://www.era.pt/pg_home.aspx?&lang=uk

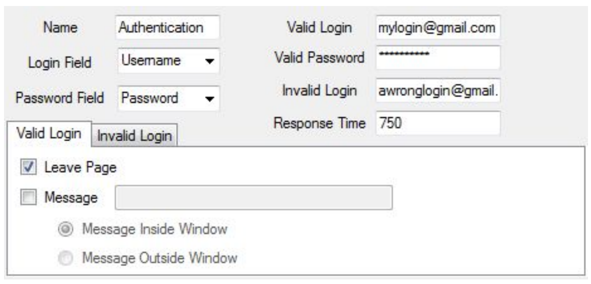
Name: Authentication
Login Field: Username
Password Field: Password
Valid Login: mylogin@gmail.com
Valid Password: **********
Invalid Login: awronglogin@gmail.
Response Time: 750
Valid Login | Invalid Login
[✓] Leave Page
[ ] Message
Message Inside Window
Message Outside Window

Figure 4 Expected Success behaviour of Authentication Pattern

automatically. In the Fig. 5 it can be seen an example of an error report where all the executed tests have passed except one. The PETTool does not give support to choose the pattern to use in each case. It purely depends on the expertise of the tester. However, the behaviour patterns are so common that it should be straightforward to identify them. The configuration steps need to be performed only once because the information gathered is saved and can be reused in following tests' configurations and in the several executions of the automatically generated tests.

B. *ERA*

In order to test the MASTER/DETAIL behaviour, the tester must start by pointing out the form under test.

The real estate agency ERA has two instances of the MASTER/DETAIL pattern: (i) District and Commune and (ii) Commune and Parish.

The tester must point out the corresponding Master field and corresponding Detail field for each pattern instance. The tool allows defining three different kinds of tests as explained in section III. In table II there is an example of a test suite that can be defined for the District and Commune MASTER/DETAIL pattern.

When the tests are run, PETTool sets the Master to the specified value and then checks the Detail value. If the Detail obeys to the defined restrictions the test succeeds, otherwise an error is reported.

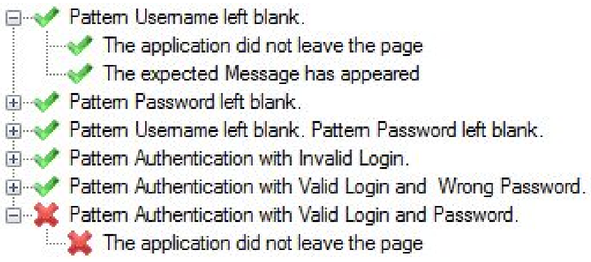
✓ Pattern Username left blank.
  ✓ The application did not leave the page
  ✓ The expected Message has appeared
✓ Pattern Password left blank.
✓ Pattern Username left blank. Pattern Password left blank.
✓ Pattern Authentication with Invalid Login.
✓ Pattern Authentication with Valid Login and Wrong Password.
✗ Pattern Authentication with Valid Login and Password.
  ✗ The application did not leave the page

Figure 5 Test Results

TABLE II TESTS FOR MASTER/DETAIL PATTERN

Tests		
District	***Constraint type***	***Commune***
Porto	Contains	Matosinhos
Lisboa	Does not Contain	Porto
Ilha do Corvo	Contains Only	Corvo

In this case study, there is a composite relation between the two instances of the Master/Detail pattern. The values that can be tested in the Commune and Parish Master/Detail pattern depend on the values set for the District field in the District and Commune Master/Detail pattern. For instance, if the tester wants to check if a Commune called Matosinhos contains a Parish called Lavra, PETTool needs to set the District to Porto in order to be able to set the Commune to Matosinhos. The PETTool allows to define composite relations between Master/Detail patterns and, when this happens, the tool tries to find automatically the value for the first Master field that allows testing the other instance of the Master/Detail pattern. This process is recursive so, after gathering the information related to the composition among several Master/Details the PETTool finds a way to test them in sequence.

VI. RELATED WORK

Model Based Testing (MBT) has been widely investigated for API testing (e.g., [3], [4]), and therefore MBT-based approaches are more common for API than for GUI testing. However, approaches applying MBT for GUI testing do exist, e.g., Memon's work [1], [5], and Paiva's work [9]. They differ in the kind of model they use and in the coverage of the test criteria used to guide the test case generation process.

The tool developed by Memon (GUITAR) generates test cases from an Event Flow Graph (EFG) model. Nodes in the EFG represent events, and directed edges represent the event-flow relationship between two events. An edge between events e1 and e2 indicates that event e2 may be invoked immediately after event e1. Concerned with the effort required to construct the EFG, Memon developed a GUI ripping tool to extract the EFG from an existing GUI [5]. In his following work [1], Memon generates a subgraph of the EFG by removing nodes and edges that are not observed in the usage information obtained from the application's real users. Then, the subgraph is augmented with probabilistic info (PEFG) in each node (event) that describes the occurrence probability of an event. Test cases are generated as event sequences that contain (i) at least one highly probable event, (ii) sequences that consider the probability of a whole path or (iii) sequences that traverse the least likely paths in the PEFG to reveal rarely-encountered faults that otherwise would be difficult to find.

The GUI Mapping tools developed by Paiva [9] is an extension of the model-based testing tool Spec Explorer[7], developed by Microsoft Research. The GUI model is written in Spec# with state variables to model the state of the GUI and methods to model the user actions on the GUI. Spec Explorer generates a finite-state machine (FSM) by exploration of the Spec# model and then test cases are generated from the FSM according to coverage criteria like full transition coverage. To run tests automatically over a GUI some additional (in-

7 http://research.microsoft.com/en-us/projects/specexplorer/

termediate) code is needed to simulate the user actions on interactive GUI controls. The GUI Mapping Tool generates such code automatically based on the mapping between model actions and GUI controls where corresponding real actions occur. Although the intermediate code is generated automatically, Paiva states that the effort needed for the construction of Spec# GUI models is too high. Following this work, she also developed a GUI reverse engineering process to extract a preliminary model from an executable GUI. This model is completed afterwards and validated in order to generate test cases. Another attempt to reduce the time spent with GUI model construction was described in [7] where a visual notation (VAN4GUIM) is designed and translated to Spec# automatically. The aim was to have a visual front-end that could hide formalism details from testers.

Both MBT and pattern-based testing, which we present in this paper, aim at reducing the effort required to construct a behavioural GUI model. Although both approaches can be augmented with already available models such models are difficult to reuse because they are frequently application dependent due to so many tiny details that can vary between GUIs - even for GUIs implementing the same behaviour. In our current approach, we tried to capture the common behaviour while maintaining the possibility to configure small variances in order to increase the degree of reuse and, in consequence, diminish the GUI modeling effort. Therefore, our approach reduces the effort of creating a model as well as the effort of creating/generating test cases.

VII. Conclusions & Future Work

GUIs have brought considerable benefits to the end-users: the interface design follows certain standards, thus reducing the users' learning curve for using new software applications. Although the sophistication of GUIs hides the underlying complexity from the user, testing GUI applications is considerably more difficult than command-line ones. Therefore, there is the need to ease software testing and to (semi-)automate the process of test generation and execution.

Exploiting the fact that the behaviour of GUIs is recurrent and following certain patterns, in this paper, we present a pattern-based approach for (semi-)automating GUI testing, dubbed PETTOOL. The approach identifies patterns of GUI behaviour and, for each one, provides a generic GUI testing solution based on known software testing techniques. Given a GUI under test and user input on what to test and how the GUI should behave, the tool automatically devises a test strategy and tests the GUI. Our proof-of-concept approach has been tested against web-based applications, such as GMail, and this paper reports our findings.

Our tests suggest that the effort required to test a GUI is not proportional to the number of tests but to the number of patterns. This way, it is possible to generate and execute a large number of tests based on a reduced number of patterns.

Since this approach is recent, there are many ways in which PETTool can be improved. The main aspect is related to the scope of the tool. There are still many patterns that can be tested. The implementation of those patterns would allow PETTool to be used in many more circumstances. GUI testing difficulties also need to be improved in PETTool. Since an error may lead to an unexpected state, in some occasions the tool cannot recover from that error. This requires an indication by the tester of how can the tool return to a well known state. However, since there might be unexpected errors, this process is not trivial. It would also be interesting to find a way to relate the multiple forms, for instance to test the authentication form and, after success, test another form automatically. This would increase the automation of the whole test execution.

References

[1] P. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE'07), Washington, DC, USA, 2007. IEEE CS.

[2] I. Burnstein. Practical Software Testing: A Process-oriented Approach. Springer Inc., 2003.

[3] A. Hartman and K. Nagin. The agedis tools for model based testing. In UML Modeling Languages and Applications, volume 3297 of Lecture Notes in Computer Science, pages 277–280. Springer, 2004.

[4] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. Modelbased Software Testing and Analysis with C#. Cambridge University Press, 2007.

[5] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering, Washington, DC, USA, 2003. IEEE CS.

[6] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In Proceedings of the International Conference on Software Engineering (ICSE'99), New York, NY, USA, 1999. ACM.

[7] R. M. L. M. Moreira and A. C. R. Paiva. Visual abstract notation for GUI modelling and testing - VAN4GUIM. In J. Cordeiro, B. Shishkov, A. Ranchordas, and M. Helfert, editors, ICSOFT (SE/MUSE/GSDCA). INSTICC Press, 2008.

[8] P. F. Oy. Ui pattern factory. http://uipatternfactory.com/, accessed in April 2010.

[9] A. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal. A model-to-implementation mapping tool for automated modelbased GUI testing. In Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM'05), pages 450–464, 2005.

[10] M. vanWelie. Welie.com - Patterns in interaction design. http://www.welie.com/patterns/index.php, accessed in May 2010.

[11] L. J. White. Regression testing of GUI event interactions. In Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96), pages 350–358, Washington, DC, USA, 1996. IEEE CS.