Model-Based Test Case Generation for Web Applications

Miguel Nabuco[1] and Ana C.R. Paiva[1,2]

[1] Department of Informatics Engineering
[2] INESC TEC
Faculty of Engineering of University of Porto
Porto, Portugal

Abstract. This paper presents a tool to filter/configure the test cases generated within the Model-Based Testing project PBGT. The models are written in a Domain Specific Language called PARADIGM and are composed by User Interface Test Patterns (UITP) describing the testing goals. To generate test cases, the tester has to provide test input data for each UITP in the model. After that, it is possible to generate test cases. However, without a filter/configuration of the test case generation algorithm, the number of test cases can be so huge that becomes unfeasible. So, this paper presents an approach to define parameters for the test case generation in order to generate a feasible number of test cases. The approach is evaluated by comparing the different test strategies and measuring the performance of the modeling tool against a capture-replay tool used for web testing.

1 Introduction

Web applications are getting more and more important. Due to the stability and security against losing data, the cloud and cloud-based web applications are progressively gaining a bigger userbase. Web applications can now handle tasks that before could only be performed by desktop applications [22], like editing images or creating spreadsheet documents.

Despite the relevance that Web applications have in the community, they still suffer from a lack of standards and conventions [23], unlike desktop and mobile applications. This means that the same task can be implemented in many different ways. Pure HTML, Javascript, Java and Flash are just some of the technologies that can be used to implement a certain feature in a web application. This makes automatic testing for Web harder, because it inhibits the reuse of test code.

However, users like to have a sense of comfort and easiness, even some familiarity, when they use their applications. For this purpose, developers use some common elements, such as User Interface (UI) Patterns [25]. In this context, UI Patterns are recurring solutions that solve common design problems. When a user sees an instance of a UI Pattern, he can easily infer how the UI is supposed to be used. A good example is the Login Pattern. It is usually composed of two

B. Murgante et al. (Eds.): ICCSA 2014, Part VI, LNCS 8584, pp. 248–262, 2014.

text boxes and a button to send the username and password data, and its function is to allow a user to access its private data.

Despite a common behavior, UI Patterns may have slightly different implementations along the Web. However, it is possible to define generic and reusable test strategies to test them. A configuration process is then necessary to adapt the test strategies to those different possible applications. That is the main idea behind the Pattern-Based GUI Testing [24] (PBGT) project in which context this research work is developed. In the PBGT approach, the user builds a test model containing instances of the before mentioned test strategies (corresponding to UI Test Patterns) for testing occurrences of UI Patterns on Web applications.

The goal of this research work is to provide different test case generation techniques from PBGT models in order to test Web applications.

The rest of the paper is structured as follows. Section 2 provides a state of the art of test generation techniques, addressing related works. Section 3 presents an overview of the PBGT approach, setting the context for this work. Section 4 presents the approach to generate and filter test cases. Section 5 presents some results and Section 6 takes some conclusions, summing up the positive points and limitations of this approach.

2 State of the Art

A good number of different test case generation techniques have been developed and researched in the past years. These techniques rely on different types of artifacts: some use software models, like Finite State Machines (FSM), others use information about the input data space, others use information from the software specifications. None of these techniques is exclusive. They can be combined to provide better results.

This paper focuses on model-based testing, which is a technique for generating a suite of test cases from models of software systems [1]. The tests are generated from these models so if they are not consistent with the software they represent, the tests will not be able to fully validate the application under test (AUT).

This testing methodology is very broad. Some approaches use axioms to build state based models using pre- and post-conditions [3]. Dick [2] developed a method to generate test cases from VDM models based on pre- and post-conditions. He created transition models where each pre- and post- condition is represented as a state. If applying a function *f()* with an argument x on a state *S1* reaches state *S2* a transition is created between states *S1* and *S2* ($S1 \rightarrow pre(f(x))$ and $post(f(x)) \rightarrow S2$). After this state machine was created, test selection techniques (such as random testing or state coverage) was allied to filter the number of tests to be performed on the AUT. Paiva [30] also uses VDM models to test user interfaces based on their specifications, represented in state transition models.

Using a slightly different approach, Claessen developed QuickCheck [4], a combinator library that generates test cases. In QuickCheck the programmer writes assertions about logical properties that a function should fulfill; these tests are specifically generated to test and attempt to falsify these assertions.

A GUI mapping tool was developed in [14], where the GUI model was written in Spec# with state variables to model the state of the GUI and methods to model the user actions on the GUI. However, the effort required for the construction of Spec# GUI models was too high. An attempt to reduce the time spent with GUI model construction was described in [15] where a visual notation (VAN4GUIM) is designed and translated to Spec# automatically. The aim was to have a visual front-end that could hide Spec# formalism details from the testers.

Algebraic specification languages have been used for the formal specification of abstract data types (ADTs) and software components, and there are several approaches to automatically derive test cases that check the conformity between the implementation and the algebraic specification of a software component [28] [29].

There are also some approaches that use FSMs. In FSM approaches, the model is formalized by a *Mealy machine*. A *Mealy machine* is a FSM whose output values are determined both by its current state and current inputs [5]. For each state and input, at most one transition is possible.

The main goal of FSM based testing tools is complete coverage. However, this can consume a lot of computational and time resources. To filter the tests to be performed, they use structural coverage criteria, such as transition coverage, state coverage, path coverage, amongst others.

Rayadurgam [8] presents a method that uses model checkers to generate test sequences that provide structural coverage of any software artifact that can be represented as a FSM.

Since most of the real software systems cannot be fully modelled with a simple FSM, most approaches use extended finite state machines (EFSM) [6]. A typical implementation on an EFSM is a state chart.

Frohlich [7] automatically creates test cases (which he defines as sequences of messages plus test data) from UML state charts. These state charts are originated from use cases which contain all relevant information, including pre- and postconditions, variations and extensions. To generate the test cases, state chart elements are mapped to a planning language.

Similar to FSMs, labeled transition systems (LTS) are also being used to generate test cases. The difference between LTS and FSM is that LTS is nondeterministic: the same trace can lead to different states. Also, the number of states and transitions is not necessarily finite and it may not have a "start" or "end" states.

There is a particular testing theory for LTS systems, called IOCO (input/output conformance) [9]. The theory assumes the AUT to be an input enabled LTS which accepts every input in every state. There are some test generation tools that implement the IOCO theory, such as TVEDA [10], TGV [11], the Agedis Tool Set [12], and TestGen [13].

An alternative approach to IOCO is *alternating simluation* in the framework of *interface automata* (IA) [16]. IA does not require input completeness of the

AUT, as opposite to IOCO. The IA approach was deeply refined and composes the foundation of Microsoft Spec Explorer Tool [17].

Some works were developed to provide test selection strategies on top of these frameworks. Feijs [18] implemented test selection based on metrics for IOCO and Nachmanson [19] implemented test selection based on graph traversal and coverage for IA.

However, for the testing of web applications, none of these approaches seemed adequate enough. The approaches mentioned require a significant effort to model the system and do not take advantage of the existence of recurrent behaviors in the application under test.

Pattern Based GUI Testing (PBGT) project aims to promote reuse by defining generic test strategies for testing common recurrent behavior on the web [26]. A big number of Web applications are built on top of UI patterns that may have slightly different implementations. PBGT defines User Interface Testing Patterns (UITP) that allows testing UI patterns and their different implementations promoting reuse and diminishing the effort of the modeling activity within a model based testing process. The description of this approach can be seen in Section 3.

3 PBGT Overview

Pattern Based GUI Testing (PBGT) is a testing approach that aims to increase the level of abstraction of test models and to promote reuse. PBGT [20] has the following components:

- **PARADIGM** — A domain specific language (DSL) for building GUI test models based on UI patterns;
- **PARADIGM-ME** — A modelling and testing environment to support the building of test models;
- **PARADIGM-TG** — A test case generation tool that builds test cases from PARADIGM models;
- **PARADIGM-TE** — A test case execution tool to execute the tests, analyze the coverage and create reports;

Currently, PARADIGM is used to test web and mobile applications, but it can be expanded to test desktop applications. By using PARADIGM-ME, can build the test model written in PARADIGM language with UI Test patterns [21]. Each UI pattern instance may have multiple configurations (for example, for a login test pattern this includes the set of usernames and passwords used, as seen in Figure 1). There are two types of configurations: Valid and Invalid. An Invalid configuration is one that simulates an erratic behavior of the user (for example, inserting wrong username and password for a Login UI test pattern, or insert letters in a text box that only accepts numbers).

Field/Value Entries	Validity	Check	Message
[password/pass; username/john_doe]	Valid	ChangePage	
[username/test; password/test]	Invalid	PresentInPage	Welcome to...

Fig. 1. Login UI test pattern configurations

A UI Test Pattern describes a generic test strategy, formally defined by a set of test goals, denoted as < Goal, V, A, C, P > where:

- **Goal** is the ID of the test;
- **V** is a set of pairs variable, inputData relating test input data (different for each configuration)with the variables involved in the test;
- **A** is the sequence of actions to be performed during test execution;
- **C** is the set of checks to be performed during test execution;
- **P** is the precondition defining the set of states in which is possible to perform the test.

In PARADIGM-ME, the tester build the web application test model, by creating the respective UITPs and joining them with connectors. These connectors define the order that the UITPs will be performed. Then, the tester has to configure each UITP with the necessary data (input data, pre-conditions, and checks). The list of supported UITPs can be seen in Table 1.

In Figure 2, a model of Tudu (an online application for managing a todo list) is shown.

Table 1. List of UITPs supported

Name	**Description**
Login	The Login UITP is used to verify user authentication. The goal is to check if it is possible to authenticate with a valid username/password and check if it is not possible to authenticate otherwise.
Find	Find UITP is used to test if the result of a search is as expected (if it finds the right set of values).
Sort	The Sort UITP is used to check if the result of a sort action is ordered accordingly to the chosen sort criterion (such as sort by name, by price and ascending or descending).
Master Detail	Master Detail UITP is applied to elements with two related objects (master and detail) in order to verify if changing the master's value correctly updates the contents of the detail.
Input	The Input UITP is used to test the behavior of input fields for valid and invalid input data.
Call	The Call UITP is used to check the functionality of the corresponding invocation. It is usually a link that may lead to a different web page.

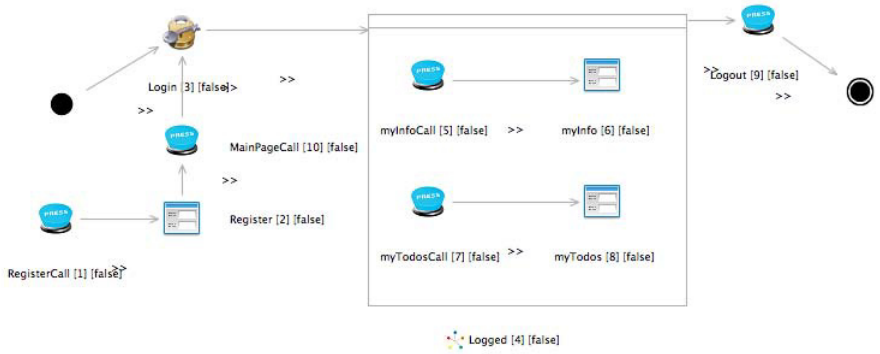

Fig. 2. Model of the web application Tudu

To perform the tests, it is necessary to establish a mapping between the model UITPs and the UI patterns of the web application to be tested. This will allow the test execution module to know which web elements to interact with that corresponds to a certain UITP described in the model. For mapping the Login UI Pattern of the Tudu model, the user must relate the login (and password) of the model with the login (and password) text box within the application under test (Figure 3) by clicking on it. PARADIGM-ME will save the following information from the user clicks:

- **Text boxes ID's** — the ID property of text boxes.
- **Images** — The image of the authentication form. This is saved through Sikuli [27] and is used when the tool is not able to identify the object by its ID.
- **Area coordinates** — The coordinates of the object. When the two previous methods fail, the tool uses the coordinates in order to interact with the web object.

After the mapping is made, PARADIGM-TG will then generate test cases from the model. The test case generation is the focus on this paper, and it is

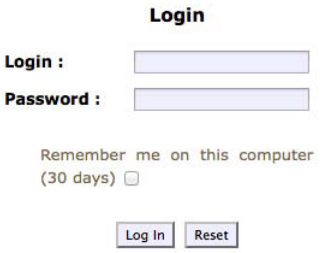

Fig. 3. Login form web application Tudu

fully described in Section 4. With the test cases generated, PARADIGM-TE will perform the tests on the web application and provide reports with the results.

4 Test Case Generation

This section explains how the test cases are generated and filtered from the web application model previously built and configured with test input data. A PARADIGM model has two mandatory nodes: an Init and an End node (where the model begin and ends respectively). Between these two nodes, there are several types of elements present: UITPs, Groups and Forms. Each UITP (which will be referred from this point onwards as "element") contains a specific ID, a number that identifies the element in the model. Forms and Groups are used to structure a model. A Group (Figure 2, Node 4, *Logged*) contains a set of UITPs that can be performed in any order and also be optional. A Form (Figure 2, Node 2, 6 and 8) contains a PARADIGM model inside it. It is used for structural reasons, to simplify the view and modeling of different sections of a web application.

4.1 Path Generator

The first step for test case generation is to flatten the model. A PARADIGM model, after being flattened, can be seen as a graph that starts in Init and finished in End and has nodes in between that are only UITP (there are no groups nor forms). So, the next step required for the test case generation is to calculate all possible paths within that graph. A path is a sequence of elements that define a possible execution trace. A model can generate many paths, based on the connectors between elements.

Considering that a model contains **x** different branches (considering the graph), **y** elements that can be performed in any order and **z** optional elements, the number of total paths generated will be:

$$TestPaths = 2^z xy! \tag{1}$$

As an example, consider a web form where the user has two mandatory inputs (ID=1 and ID=2) in any order followed by other 2 optional inputs (ID=3 and ID=4). The number of paths will then be $2^2 2! = 8$.
The sequences generated will be the following:

{[1,2], [2,1], [1,2,3], [2,1,3], [1,2,4], [2,1,4], [1,2,3,4], [2,1,3,4]}

Each of these sequences compose a Test Path, that can contains multiple test cases according to the configurations defined for each UITP.

4.2 Path Filter

In case the model generates too many test paths for the available resources or if the testing goal is to test a single component and not the whole web application,

a path filter component was implemented to reduce the number of Test Paths generated.

The path filter will restrict the number of test paths generated based on three simple conditions:

- **Cycles** — A Test Path can have cycles, i.e., a sequence of elements that can repeat itself multiple times. It is possible to reduce the number of Test Paths created by limiting the number of times each cycle is performed. In Figure 4, a cycle can be seen in the elements {[1,2]}.
- **Mandatory and Exclusion Elements** — If the tester wants to test a certain feature or a certain workflow that can only be reached following a certain path, he can explicitly define the mandatory and exclusion elements. By excluding one element (or one sequence of elements), every Test Path that contains that element will not be generated. Also, by excluding an element other elements may be automatically excluded.
- **Random** — The tester can also define a maximum number of Test Paths to be randomly generated. These Test Paths may already be filtered by the two conditions described above, so this is the last condition to be applied.

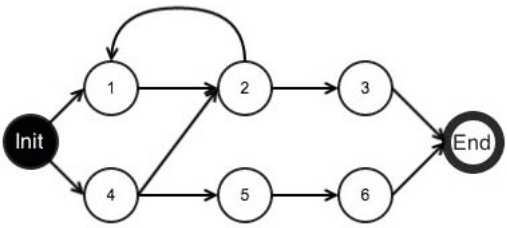

Fig. 4. Sample Test Model

4.3 Path Tree

After the initial Test Path Filtering, each Test Path must be decomposed in multiple test cases.

One element can contain multiple configurations. A Login element can contain valid and invalid configurations and each one of these can lead to different outcomes. Therefore, each configuration must generate a different test case.

Considering a Test Path with $\mathbf{x}$ elements, each with $\mathbf{y_x}$ configurations, the total number of Test Cases generated will be:

$$Tc = y_x \times y_{x-1} \times ... \times y_1 \tag{2}$$

To make the test case generation easier, a tree structure was created, as seen in Figure 5.

Login	Username	Password
ValidLoginA	admin	admin
InvalidLoginB	invalid	invalid

Sort	Type	Order
SortA	name	ascending
SortB	price	descending

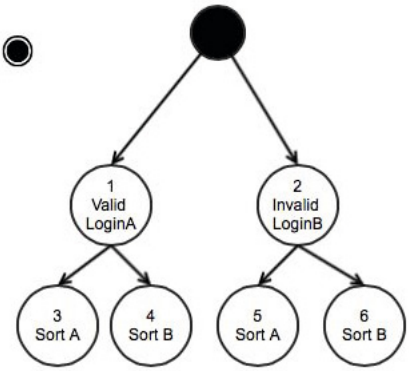

Fig. 5. PARADIGM model, configurations and the corresponding Test Tree

The tree is then traveled using a depth-first search algorithm. An iterator starts from the tree root node and follows one branch until the end. Then, it marks the last node visited, saves the branch as a test case and starts again from the beginning, never traveling into the nodes already marked as visited.

From the Figure 5, the travelling order would be:

[1,3,1,4,2,5,2,6]

Creating the following test cases:

{[1,3], [1,4], [2,5], [2,6]}

4.4 Test Case Filtering

From the example above, it can be seen that from two simple Login and Sort UI Test Patterns with two different configurations, 4 test cases are generated. If this is scaled to real-life applications, there can be a test case explosion, with too many test cases to perform regarding the time and computational resources available. Therefore, just like a filter was applied to the Test Paths, a different filter may be applied to reduce the number of test cases. The metrics applied in the filter are the following:

– **Specific configurations** — If the tree is filtered so that only test cases with specific configurations are considered, the number of test cases to be performed can drastically reduce. In Figure 5, if it is specified that only valid logins are of interest, the total number of test cases drops from 4 to 2, decreasing 50%. The modelling tool provides enough flexibility so that the tester can generate test cases for the specific components he wants to test.

– **Random** — As with the Path Filtering explained in Section 4.2, the tester can also specify a number of test cases to be randomly chosen from the total set.

4.5 Test Case Generation Modes

To apply the filters, the user can choose to do it manually (specifying for each option described previously which are the parameters), provide a text file with a set configuration or choose one of the automatic test strategies options available. The test strategies implemented are the following:

– **Default Strategy** – This strategy generates all test paths, but only performs N test cases per each test path (N is defined by the user).
– **All Invalids Strategy** – The goal of this test strategy is to test all invalid configurations of all the elements, to find erratic behaviors in the application under testing. So, this strategy generates one test case for each invalid configuration defined.
– **Random Strategy** – This strategy picks a random test case from a random test path and runs it. Then it picks a random test case again. This cycle will continue, until the user stops the program execution.

In Figure 6, the option menu to select the test strategy can be seen.

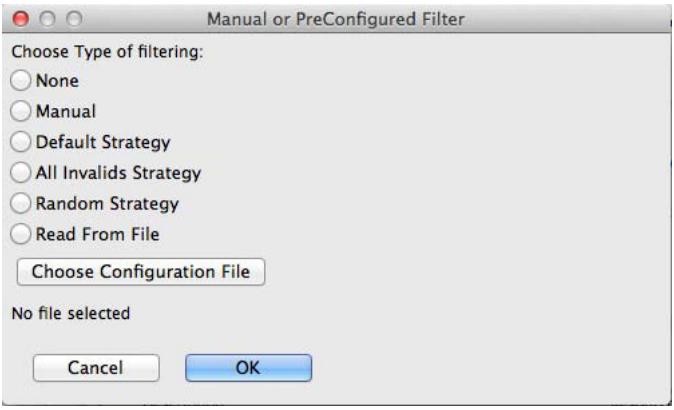

Fig. 6. Test Strategies Menu

4.6 Exporting to PARADIGM-TE

After all the filters are applied, and the test cases are selected, the tool will then export the test cases into XML files, so that the test execution tool (PARADIGM-TE) can run them and generate reports. Each Test Path (with several test cases) is stored in a different XML file. This allows an easier test execution and debug, as it is easier to know if there is any Path not correctly defined. An example of a XML file of a simple two nodes test case can be seen in Figure 7.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Script>
<Path value="[1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6]">
<Config/><Init/>
<Call check="presentInPage" flag="true" name="LinkNormal" number="1.1" optional="false"
result="yes"><Field name="call_1.1"/></Call>
<Login check="changePage" flag="false" message="" name="LoginNormal" number="1.2"
optional="false" validity="Valid"><Value field="username_1.2" fieldName="username"
value="user"/><Value field="password_1.2" fieldName="password" value="pass"/></Login>
</Path></Script>
```

Fig. 7. Two nodes XML file

5 Evaluation

To evaluate this approach, two different experiments were made: one to see the results of using different test strategies and the other to see how efficient this approach could be against another web testing tool.

The experiment was performed over three different open-source web applications: **Tudu** (Figure 2), a web application for managing todo lists; **iAdressBook**, a PHP address book application; and **TaskFreak**, a task management web application. Each model contained several configurations for each element.

To measure the effectiveness of the approach, each web application source code was injected with mutants. Each mutant would change boolean and arithmetic operations and conditions (i.e., change a *true* to *false*, or a *greater than* for a *smaller or equal than*).

5.1 Test Strategies

The first experiment compared the effectiveness of the three automatic test strategies implemented: *Default* (with N=1), *All Invalid* and *Random*.

The results of this experiment in the web application TuDu (with 35 mutants) can be seen in Table 2. Comparing the *Default* strategy with the *Invalids* one, it was found that *Invalids* killed most of the mutants *Default* could kill, but it took more test cases to do so. However, the *Invalids* could kill some mutants that *Default* could not kill. These were mutants that affected the code of the authentication methods (the user could login with an invalid login and password) or incorrect data input verification.

Comparing the *Default* strategy 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

Table 2. Comparison between different test strategies

	Default	**Random**	**Invalids**
Number of mutants killed	24	32	27
Test Cases to kill one mutant (avg)	1.3	5.2	3.7
Time (min) per mutant (avg)	9.1	26.5	18.5

test cases to kill them. Also, since it is totally Random, it repeated some test cases, therefore 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.

5.2 Comparison against a Capture-Replay Tool

The goal of the second experiment was to compare how many mutants PARADIGM and a capture-replay tool would kill. The capture-replay tool chosen for this experiment was Selenium-IDE. Selenium was chosen because PARADIGM uses Selenium libraries in its core, and so it makes sense to see what advantages could this approach bring over regular Selenium testing. Both of the tools 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 (in the case of PARADIGM, the model). The test strategy chosen was the default with N=1 (one test case per test path, as explained in Section 4.5).

To perform the experiment, a set of test goals were delivered to two different testing teams (one for PARADIGM and other for Selenium), that had no interaction with each other. Both had to configure the application, run the tests, and measure the results.

The effort required to build and configure the application scripts (for Selenium) and the models (for PARADIGM) can be seen in Table 3. The effort to build the application models in PARADIGM is greater than to create test scripts in Selenium. However, in case the web application layout changes, it is easier to change the model in PARADIGM than to change the scripts in Selenium.

The applications were then tested with both tools and the results can be seen on Table 4.

PARADIGM tool managed to successfully kill 82.7% of the mutants, when Selenium killed 72.3%.

The extra effort needed to model the application (as opposed to the *crawling* nature of Selenium, in which the testers simply interact with the application and their steps are recorded), compensates in the number of mutants killed.

Selenium uses scripts to run the tests, which limits the range of inputs and checks. If a tester wants to add a different set of input and check data, he has to create different script. When the tester creates the model in PARADIGM, it is

Table 3. Effort (in minutes) required to build and configure models/scrips

	PARADIGM	**SELENIUM**	**Difference**
TuDu	73	60	+13
TaskFreak	164	101	+63
iAdressBook	137	75	+62

Table 4. Evaluation results

	Mutants	**PARADIGM**	**SELENIUM**
TuDu	28	24	16
TaskFreak	76	59	53
iAdressBook	69	60	56
Total	**173**	**143(82.7%)**	**125(72.3%)**

easy to continuously add different input data and checks, since the model stays the same. With more configurations, the number of test cases increases and the application under test is more tested.

Also, while the Selenium team created an average of 5 to 7 scripts to test each web application according to the test goals defined, the PARADIGM tool could generate hundreds of different test cases from each model (for TuDu alone, 245 test cases were generated).

6 Conclusions and Future Work

This paper proposed a test generation and filtering technique for model-based testing of Web applications. The models contain information in the form of UI Test Patterns linked with connectors. Each UI Test Pattern contains specific configurations with the data needed for test execution.

Considering that test cases cannot run forever, this paper presented several filters that were applied to provide flexibility and reduce the number of test cases generated. Firstly the filters were applied to the Test Paths and then to each Test Path element (an UI Test Pattern) configurations. Some test strategies were then created, each with its own broadness and coverage.

This approach was tested on three different web applications. The several test strategies were compared between themselves. Also, this approach was compared with the same set of tests performed with a capture-replay tool. PARADIGM proved to be effective, as the filtering can provide better coverage, finding more bugs and killing more mutants.

In general, it is possible to see clear advantages of this approach:

- **Distributing Test Suites** — Since the test process can be configured in order to generate different test suites, it is possible to spread different test suites through different workstations and run tests concurrently and in less time.
- **Easier debugging** — If a critical error occurs that may stop the execution of the testing program altogether, it is easy to see which element or configuration caused that error. All the information is stored so that the user can replay the steps even manually. Also, as the results are updated after each Test Path, there is no need to run all the selected tests again in case of error.

The features planned for the near future include a broader set of filtering options, test strategies and test coverage statistics.

Acknowledgments. This work is financed by the ERDF — European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by 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.

References

1. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-based testing in practice. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), pp. 285–294. ACM, New York (1999)
2. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Larsen, P.G., Wing, J.M. (eds.) FME 1993. LNCS, vol. 670, pp. 268–284. Springer, Heidelberg (1993)
3. Katsiri, E., Mycroft, A.: Model checking for sentient computing: an axiomatic approach. In: Proceedings of the First International Workshop on Managing Context Information in Mobile and Pervasive Environments (SME 2005), CEUR-WS, Ayia Napa (May 2005)
4. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (eds.) Proc. of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 268–279. ACM (2000)
5. Mealy, G.H.: A Method for Synthesizing Sequential Circuits. Bell Systems Technical Journal 34, 1045–1079 (1955)
6. Cheng, K.-T., Krishnakumar, A.S.: Automatic Functional Test Generation Using The Extended Finite State Machine Model. In: 1993 30th Conference on Design Automation, June 14-18, pp. 86–91 (1993)
7. Fröhlich, P., Link, J.: Automated Test Case Generation from Dynamic Models. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, p. 472. Springer, Heidelberg (2000)
8. Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: Proceedings. Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems-ECBS, pp. 83–91 (2001)
9. Tretmans, J.: Test Generation with Inputs, Outputs, and Repetitive Quiescence. Software-Concepts and Tools 17, 103–120 (1996)
10. Phalippou, M.: Relations d Implantation et Hypotheses de Test sur des Automates a Entrees et Sorties. PhD thesis, L Universite de Bordeaux I, France (1994)
11. Jard, C., Jeron, T.: TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. Software Tools for Technology Transfer 7(4), 297–315 (2005)
12. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. In: Int. Symposium on Software Testing and Analysis - ISSTA 2004, pp. 129–132. ACM Press, New York (2004)
13. He, J., Turner, K.: Protocol-Inspired Hardware Testing. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) Int. Workshop on Testing of Communicating Systems, vol. 12, pp. 131–147. Kluwer Academic Publishers (1999)

14. Paiva, A.C.R., Faria, J.C.P., Tillmann, N., Vidal, R.A.M.: A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 450–464. Springer, Heidelberg (2005)
15. Moreira, R.M.L.M., Paiva, A.C.R.: Visual Abstract Notation for Gui Modelling and Testing - VAN4GUIM. In: ICSOFT 2008, pp. 104–111 (March 4, 2008)
16. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC / SIGSOFT FSE, pp. 109–120 (2001)
17. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with spec explorer. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 542–547. Springer, Heidelberg (2005)
18. Feijs, L.M.G., Goga, N., Mauw, S., Tretmans, J.: Test selection, trace distance and heuristics. In: Proceedings of the IFIP 14th International Conference on Testing Communicating Systems (TestCom 2002), pp. 267–282. Kluwer (2002)
19. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. In: Avrunin, G.S., Rothermel, G. (eds.) Proc. of the 2004 ACM SIGSOFT International Sym- posium on Software Testing and Analysis (ISSTA 2004), pp. 55–64 (2004)
20. Moreira, R., Paiva, A., Memon, A.: A Pattern-Based Approach for GUI Modeling and Testing. In: Proceedings of the 24th annual International Symposium on Software Reliability Engineering, ISSRE 2013 (2013)
21. Welie, M., Gerrit, C., Eliens, A.: Patterns as tools for user interface design. In: Workshop on Tools for Working With Guidelines, Biarritz, France (2000)
22. Garrett, J.J.: Ajax: a new approach to Web applications (2006), http://www.adaptivepath.com/publications/essays/archives/000385.php
23. Constantine, L.L., Lockwood, L.A.D.: Usage-centered engineering for Web applications. IEEE Software Journal 19(2), 42–50 (2002)
24. Monteiro, T., Paiva, A.: Pattern Based GUI Testing Modeling Environment. In: 4th International Workshop on Testing Techniques & Experimentation Benchmarks for Event-Driven Software, TESTBEDS 2013 (2013)
25. Nabuco, M., Paiva, A., Camacho, R., Faria, J., Inferring, U.I.: Patterns with Inductive Logic Programming. In: 8th Iberian Conference on Information Systems and Technologies (2013)
26. Cunha, M., Paiva, A., Ferreira, H., Abreu, R.,, P.: A Pattern-Based GUI Testing Tool. In: 2nd International Conference on Software Technology and Engineering (ICSTE 2010), pp. 202–206 (2010)
27. Sikuli API (last acessed February 2014), https://code.google.com/p/sikuli-api/
28. Andrade, F.R., Faria, J.P., Paiva, A.: Test generation from bounded algebraic specifications using alloy. In: ICSOFT 2011, 6th International Conference on Software and Data Technology (January 2011)
29. Rebello de Andrade, F., Faria, J.P., Lopes, A., Paiva, A.C.R.: Specification-driven unit test generation for java generic classes. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 296–311. Springer, Heidelberg (2012)
30. Paiva, A.C.R., Faria, J.P., Vidal, R.M.: Specification-based Testing of User Interfaces. In: Proceedings of the 10th DSV-IS Workshop - Design, Specification and Verification of Interactive Systems, Funchal, Madeira, de Junho 4-6 (2003)