Towards automated load testing through the user interface[*]

Bruno Teixeira and José Creissac Campos[0000−0001−9163−580X]

University of Minho & INESC TEC, Braga, Portugal
jose.campos@di.uminho.pt

Abstract. Slight variations in user interface response times can significantly impact the user experience provided by an interface. Load testing is used to evaluate how an application behaves under increasing loads. For interactive applications, load testing can be done by directly calling services at the business logic or through the user interface. In modern web applications, there is a considerable amount of control logic on the browser side. The impact of this logic on applications' behaviour is only fully considered if the tests are done through the user interface. Capture reply tools are used for this, but their use can become costly. Leveraging an existing model-based testing tool, we propose an approach to automate load testing done through the user interface.

Keywords: Model-based testing, load testing, capture and replay

1 Introduction

Software testing [9] aims to increase confidence in the quality of a piece of software. Load testing is a particular type of software testing, which aims at testing a system's response under varying load conditions by simulating multiple users accessing the application concurrently.

Load testing can be split into API and UI (User Interface) load testing. API load testing is done by directly calling the services at the business layer level. This avoids the complexity, and work load, of automating the interaction with the application's user interface. Although strategies can be used to create combinations of different types of request, side stepping the UI risks making the tests less representative of actual use. Tests that ignore the user interface do not take into account the control logic programmed into the browser. This logic can range from dialogue control, which can help mask or exacerbate delays

[*] This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

Author's version of the paper published in Human-Computer Interaction, volume 14143 of Lecture Notes in Computer Science, pages 514-522. Springer. 2023. The final version is available at Springer via: http://dx.doi.org/10.1007/978-3-031-42283-6_28.

in the access to back-end services, to the full business logic of the application, depending on the type of web application under test. In any case, the impact of, at least, part of the application logic on performance is not assessed. This is relevant because small variations in user interface response times can have a huge impact on their user experience [4].

UI load testing addresses this by interacting with the browser to simulate user interaction. This type of test is expensive to setup, usually requiring a capture phase where use behaviour is recorded for later replay during testing.

In this paper, we present an approach to automate UI load testing of web applications. This work is part of an ongoing effort to explore the use of model-based testing to automate the testing of user interfaces.

2 Background

Load tests are non-functional tests, aimed at determining how well a system performs under high work loads. In the web applications' context, load testing involves the use of tools to simulate the execution of the application when subjected to a specific workload, and the analysis of measurements according to predefined benchmarks. These benchmarks cover metrics such as the number of virtual users, throughput, errors per second, response time, latency and bytes per second [6]. The goal is to identify performance bottlenecks in order to prevent end-users from encountering any problems during peak load.

The need to automate the testing process is essentially due to the need to repeat the same tests, as well as the need to increase test coverage. Model-based testing (MBT) [13] is a black-box testing technique that supports the automation of software testing, from test generation to test results' analysis [12,1]. This method compares the state and behaviour of a product (the system under test – SUT) with an abstract model (the oracle) that represents the behaviour of the SUT. Discovery of system errors is achieved by comparing the results of the tests in the SUT with the *predictions* of the oracle, in order to detect inconsistencies between both. MBT's main limitations are connected to the need to develop and maintain the oracle, and the potential for an explosion of test cases during the generation process. We chose to explore MBT due to its potential to automate the testing process from a model of the SUT.

Several authors have explored the application of MBT techniques to Graphical User Interfaces (GUI) (examples include [7,8,10], but see also [11] for a review). It should be noted, however that the focus has been product quality, much more than quality in use (cf. the ISO/IEC 25010 standard [5]). This is to be expected, since the oracle typically captures functional requirements of the SUT. An example of folding usage considerations into an MBT process is presented in [2]. The paper is part on an ongoing effort to explore how MBT can be used to automate the testing of user interfaces. In the context of this effort we have developed the TOM tool [10]. Here we look at how the tool might be used to perform UI load testing.

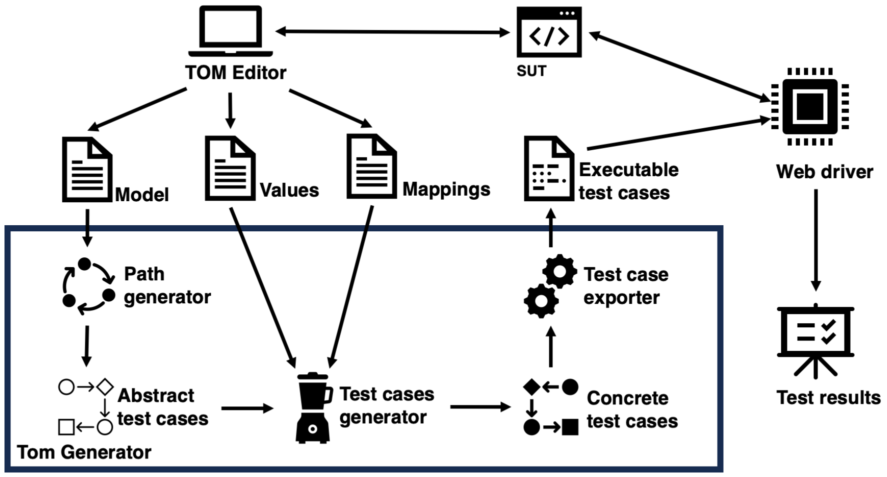


Fig. 1. The TOM framework

3 Design of the proposed approach

In order to support automated model-based load testing from the GUI, assuming a model of the SUT is available, four requirements were identified: (Step 1) generation of multiple similar tests from the model; (Step 2) automation of the interaction with the browser; (Step 3) load simulation on the application by running multiple tests concurrently; (Step 4) capture of performance metrics and generation of reports for results' analysis.

The TOM framework (see Figure 1) already provides initial solutions for the first two steps. Its test cases generation component (TOM Generator) receives as input a model of the application's GUI (a state machine, represented in XML, where states represent windows or pages of the interface), and generates executable tests in Java using Selenium WebDriver (Step 1). Models are abstract representations of the interface, that is, the state machine does not identify the concrete elements of the implementation. TOM performs both the step of generating abstract tests over the model and refining them to concrete (executable) ones. For this, it needs two additional inputs (a *Mapping* file and a *Values* file).

The *Mapping* file identifies which implementation elements correspond to which logical (abstract) elements in the model. It consists of a JSON object defining a mapping between the model and the elements present in the HTML page. The *Values* file defines concrete values to be used when creating executable tests. As mentioned above, interaction with the browser (Step 2) is achieved through Selenium WebDrive, a remote control interface that enables programmatic introspection and control of the browser.

The next step (Step 3) is to test the SUT. We divide this into two sub-steps: a functional testing phase to guarantee that all test cases are successful, followed by a load testing phase. A solution to run tests concurrently is needed as, in the available version, TOM can only run tests sequentially. Apache JMeter[1] is used to achieve this. However, since the goal is to perform load tests through the application's UI, it is not possible to use the traditional HTTP Request component that JMeter offers, as this type of component is used to make requests to the application's API. Thus, it was necessary to explore alternatives supporting the execution of tests written using Selenium WebDriver. These are called samplers. JMeter features several types of samplers, and after analysis, the option was to use the JUnit Request sampler.

In the last step of the process (Step 4), it is necessary to evaluate and analyse the results of the test cases' execution. As the main goal is to analyse the effect of load on the application, it is relevant to analyse performance metrics. However, as mentioned above, there is a first testing phase where each test is executed separately (to find possible failing tests). Besides supporting functional tests, this guarantees that only non-failing tests are performed in the load testing phase in order not to waste resources and time. This is relevant since load testing requires a large amount of resources and computing power. After analysis of alternatives, the Allure framework was adopted to present the results of the functional testing phase. Regarding the analysis of the load testing phase, JMeter provides reports on this aspect.

4 Implementation

The TOM framework is composed of three components: TOM Generator, TOM Editor and TOM App. TOM Generator (see Figure 1) contains the core of the TOM Framework. This component has a modular architecture to ensure the adaptability of the framework and includes modules to read and translate different types of models to their internal representation, as well as the modules to generate, transform and execute the test cases. The TOM Editor (see Figure 3, right) is a browser extension that is responsible for developing GUI models. The editor supports capturing the user's interaction and defining the system model based on that interaction. The TOM App (see Figure 2) consists of the presentation layer of the TOM Framework. It supports the process of generating and executing test cases.

A number of changes had to be made to the existing TOM framework implementation in order to support the execution of load tests. These were done both on the framework's back-end (TOM Generator) and front-end (TOM App).

4.1 Back-end – TOM Generator

Two main changes were made in the back-end. One was the implementation of the Headless Mode property, the other the implementation of an option to generate random data.

Headless mode is a process of executing tests in browsers without displaying the user interface. This means that the HTML page that is under test is not

[1] https://jmeter.apache.org/

rendered on the screen (which should save time during test execution). This mechanism helps run multiple concurrent tests in a single machine, or in remote machines. Without it, multiple browser instances would have to be concurrently opened and closed, which is likely to have a significant impact on the time needed to run the tests. This mode is optional to cater for situations where actually displaying the pages is considered relevant (for example, when the GUI's rendering times need to be considered). Implementing it, meant changes mostly to the test cases generation process, which is responsible for converting abstract tests into concrete test cases.

The other change was the implementation of an option to generate random data. Originally, the TOM Generator allowed the filling of forms with the data directly present in the Values file. This becomes a problem when performing load tests, as we have to execute the same test multiple times. Repeatedly using the same values will, in many cases, fail (consider the task of user registration). This problem was solved by adding a property to the Mapping file that tells TOM Generator to generate random data (based on the value provided). This prevents exactly identical executable tests. The code needed to read the configuration files was updated to process the new attributes added to the mapping file, in particular, related to the generation of random values.

A new exporter was created to generate the load tests. This was done starting from a generic test template in JMX (a JMeter format) to which the tests are added. Routes to support access to the new functionalities were also implemented.

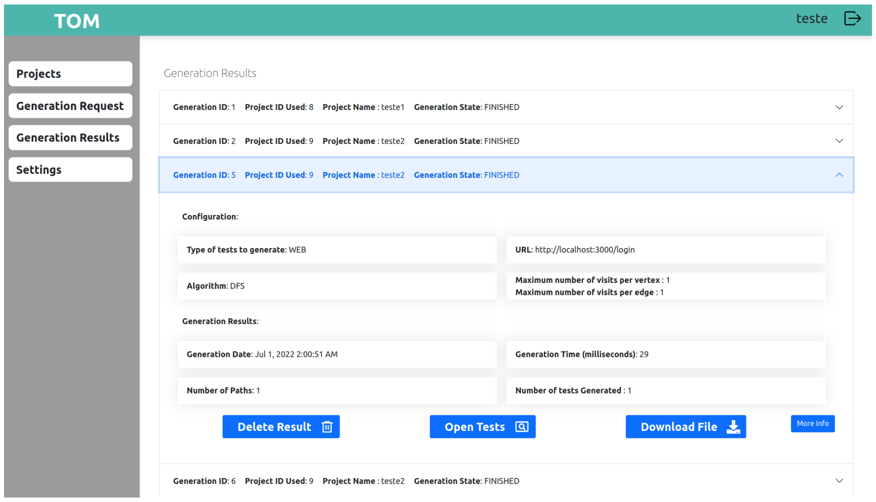
4.2 Front-end – TOM App

A new version of the TOM framework front-end (the TOM App) was developed to support the new functionalities. The UI of the application is divided in three main tabs (see Figure 2): Projects, Generation Requests, and Generation Results.

The Projects tab contains all the functionalities for managing projects. Users can view the list of their projects and add new projects. Additionally, users can observe all the components that make up the project, such as the System Model, the Mapping file and the Values file. Deleting the project and downloading it are also available options.

The generation of test cases is available in the Generation Requests tab. This process involves three phases. In the first phase, global configurations are set, such as the URL for which the test cases will be generated, whether to use headless mode, and the graph traversal algorithm to be used to generate test cases. In the second phase, the project that will be used to generate the tests is selected (the UI model, etc.). At this stage, the user may also visualise all the configuration files that constitute the project. Finally, in the third phase, the user is presented with all the information about the test generation request and the option to execute it (which sends it to TOM Generator).

In the Generate results tab (see Figure 2) the results of previous test case generation requests are presented. Regarding each result, it is possible to visualise properties such as the number of tests generated and the time needed to



Generation Results

Generation ID: 1 **Project ID Used**: 8 **Project Name** : teste1 **Generation State**: FINISHED

Generation ID: 2 **Project ID Used**: 9 **Project Name** : teste2 **Generation State**: FINISHED

Generation ID: 5 **Project ID Used**: 9 **Project Name** : teste2 **Generation State**: FINISHED

Configuration:

Type of tests to generate: WEB
URL: http://localhost:3000/login
Algorithm: DFS
Maximum number of visits per vertex : 1
Maximum number of visits per edge : 1

Generation Results:

Generation Date: Jul 1, 2022 2:00:51 AM
Generation Time (milliseconds): 29
Number of Paths: 1
Number of tests Generated : 1

Generation ID: 6 **Project ID Used**: 9 **Project Name** : teste2 **Generation State**: FINISHED

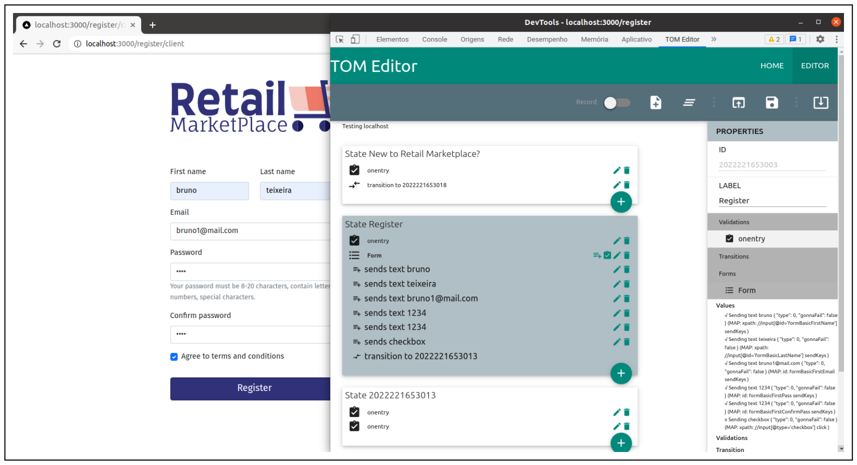
Fig. 2. TOM App: Generation Results

generate the tests, as well as download the corresponding files to run the tests at a later date and in an appropriate setup, or open the test set. Opening a particular test set provides access to the execution of the tests. This is done in two phases as explained above.

5 Applying the framework

We now briefly describe two applications of the framework. The first is its application to a preexisting e-commerce platform (Retail Marketplace) developed in the scope of a MSc. curricular unit by the first author and colleagues.

The concrete functionality tested herein consists of the process of registering a buyer. Users must initially navigate to the login form, where they find the option ”register new user”. Then, they must chooses the option ”register as purchaser”, and finally fill in the registration form on the platform (see Figure 3, left). The model of the user interface was developed using the TOM Editor, which partially automates the process using capture-replay. A test was generated to exercise the registration process. Originally, this test consisted of registering a user with name “Bruno Teixeira” using the email “bruno@mail.com” and password “1234”. This test passed successfully when run on its own (first stage of testing). Load testing consisted of simulating 20 users performing the registration process simultaneously. For testing purposes the test was run locally, and configured to use headless mode. As required for load testing, random values were used for the inputs. As can be seen at the top of Figure 4, from the 20 tests executed, two failed. This failure may be related to a delay in receiving the reply from the Retail Marketplace app, which meant that subsequent attempts to locate elements in the interface failed. While this in itself is an indication of degraded



First name: bruno
Last name: teixeira
Email: bruno1@mail.com
Password: ••••
Your password must be 8-20 characters, contain letter numbers, special characters.
Confirm password: ••••
☑ Agree to terms and conditions

State New to Retail Marketplace?
onentry
transition to 2022221653018

State Register
onentry
Form
sends text bruno
sends text teixeira
sends text bruno1@mail.com
sends text 1234
sends text 1234
sends checkbox
transition to 2022221653013

State 2022221653013
onentry
onentry

ID: 2022221653003
LABEL: Register
Validations: onentry

```
√ Sending text bruno { "type": 0, "gonnaFail": false } (MAP: xpath: //input[@id='formBasicFirstName'] sendKeys )
√ Sending text teixeira { "type": 0, "gonnaFail": false } (MAP: xpath: //input[@id='formBasicLastName'] sendKeys )
√ Sending text bruno1@mail.com { "type": 0, "gonnaFail": false } (MAP: id: formBasicFirstEmail sendKeys )
√ Sending text 1234 { "type": 0, "gonnaFail": false } (MAP: id: formBasicFirstPass sendKeys )
√ Sending text 1234 { "type": 0, "gonnaFail": false } (MAP: id: formBasicFirstConfirmPass sendKeys )
x Sending checkbox { "type": 0, "gonnaFail": false } (MAP: xpath: //input[@type='checkbox'] click )
```

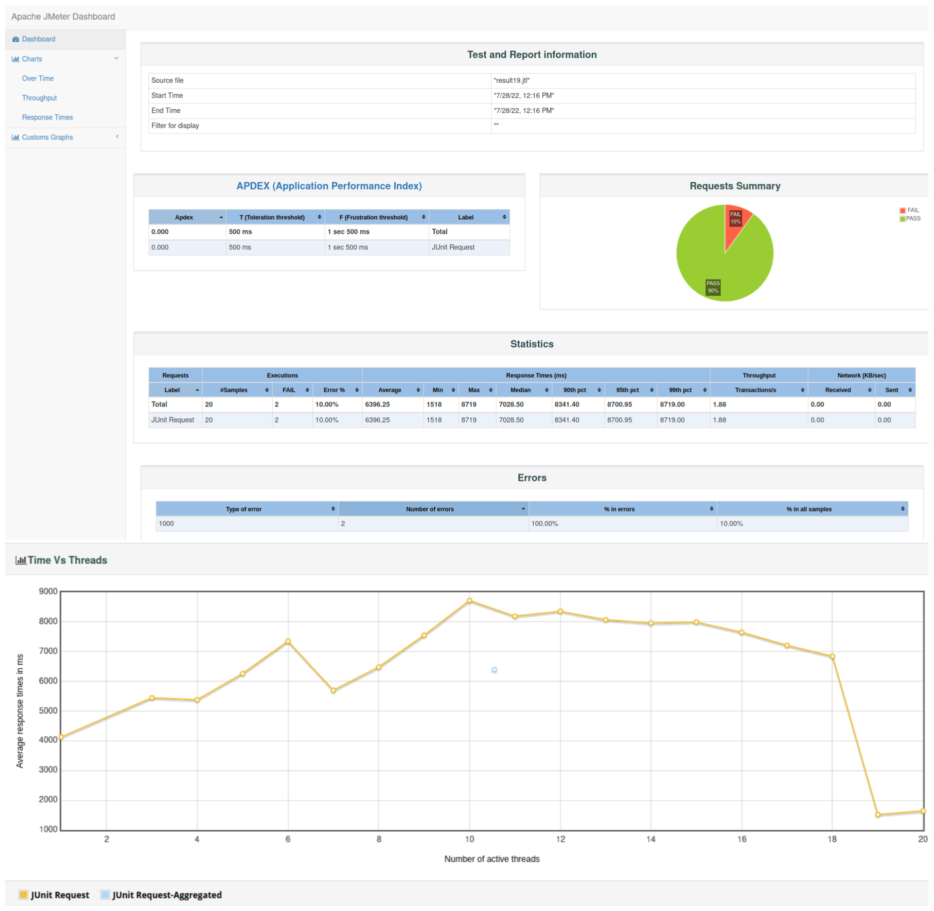
Fig. 3. The Retail Marketplace registration form (left) and TOM Editor (right) – from [3]

performance (indeed performance degrades substantially with 18 threads), in the future, the possibility of defining the pace of the interaction should be supported.

Another example of use is the application of the framework to the analysis of the TOM app itself. This involved the creation of a model of the app. The model consisted of 23 states, and 297 tests were generated (in 556ms). All tests passed the first stage. For the second phase, 11 tests were selected for execution. The tests were run on remote machines using the Google Cloud platform. Several scenarios were attempted, with the goal of assessing the impact of headless mode and the impact of performing the test through the UI versus direct back-end API calls. It was possible to conclude that using headless mode provided some level of improvement in terms of time required to run the tests, although not as much as we were expecting – on average a time saving of 5.7%. The main conclusion, however, was that the computing power required to perform realistic tests (in terms of number of concurrent accesses) increases quite considerable when running the test through the UI, even in headless mode. In an environment with 32 virtual CPUs, each with 138 MB of RAM, it was not possible to run the tests for 1500 threads in the UI case, while in the back-end API case it was possible to reach more than 30000 threads. In the future, strategies to decreases the resources demand of the framework should be researched.

6 Conclusions and future work

We have described a model based approach to automate the load testing of web applications that takes into consideration their user interface layer. The short term goal has been to support the automated generation and execution of more realistic load tests. Web application are increasingly using client side code and



Test and Report information

Source file	"result19.jtl"
Start Time	"7/28/22, 12:16 PM"
End Time	"7/28/22, 12:16 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.000	**500 ms**	**1 sec 500 ms**	**Total**
0.000	500 ms	1 sec 500 ms	JUnit Request

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**20**	**2**	**10.00%**	**6396.25**	**1518**	**8719**	**7028.50**	**8341.40**	**8700.95**	**8719.00**	**1.88**	**0.00**	**0.00**
JUnit Request	20	2	10.00%	6396.25	1518	8719	7028.50	8341.40	8700.95	8719.00	1.88	0.00	0.00

Errors

Type of error	Number of errors	% in errors	% in all samples
1000	2	100.00%	10.00%

Fig. 4. JMeter reports

testing the back-end APIs directly does not take into consideration this code. However, creating UI load tests that interact with the application through the user interface is time consuming. By using a model-based approach we were able to considerably automate the process. One pending problem is the high level of computational resources required to run the tests. What level of resources is needed to run the approach effectively, and whether the resources requirements can be decreased, will have to be further investigated.

The long term goal is to support the consideration of user experience criteria when load testing web applications. Small differences in UI reaction time can have a great impact on user experience. How the user interface uses the back-end services can help mask or exacerbate delays created by the back-end. It can also create delays in the user interface itself. This means that from a user experience perspective, the impact of load on the application must be measured at the user interface. In future work we plan to explore how the approach proposed herein can be used to assess these aspects, and also how the approach compares to API load testing for different classes of applications. Web applications with different levels of logic on the client side (the browser) will be tested and the results compared with tests run directly on their business logic APIs.

References

1. Busser, R.D., Blackburn, M.R., Nauman, A.M.: Automated model analysis and test generation for flight guidance mode logic. In: 20th DASC. 20th Digital Avionics Systems Conference (Cat. No. 01CH37219). vol. 2, pp. 9B3–1. IEEE (2001)
2. Campos, J., Fayollas, C., GonÃ§alves, M., Martinie, C., Navarre, D., Palanque, P., Pinto, M.: A "more intelligent" test case generation approach through task models manipulation. Proceedings of the ACM on Human Computer Interaction **1**(EICS), 9:1–9:20 (Jun 2017). https://doi.org/10.1145/3095811
3. Gonçalves, M.J.R.: Model-based Testing of User Interfaces. Msc. dissertation, Escola de Engenharia, Universidade do Minho (2017)
4. Gray, W.D., Boehm-Davis, D.A.: Milliseconds matter: an introduction to microstrategies and to their use in describing and predicting interactive behavior. Journal of experimental psychology. Applied **6**(4), 322–335 (2000). https://doi.org/10.1037//1076-898x.6.4.322
5. ISO/IEC: ISO/IEC 25010:2011 systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Organization for Standardization (2011), `https://www.iso.org/standard/35733.html`
6. Jain, R.: The Art of Computer Systems Performance Analysis. Wiley (1991)
7. Memon, A., Banerjee, I., Nagarajan, A.: GUI ripping: Reverse engineering of graphical user interfaces for testing. In: 10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings. pp. 260–269. Citeseer (2003)
8. Moreira, R.M., Paiva, A.C.: PBGT tool: an integrated modeling and testing environment for pattern-based gui testing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 863–866 (2014)
9. O'Regan, G.: Concise Guide to Software Testing. Springer (2019)
10. Pinto, M., Gonçalves, M., Masci, P., Campos, J.: TOM: a model-based gui testing framework. In: Formal Aspects of Component Software. Lecture Notes in Computer Science, vol. 10487, pp. 155–161. Springer (2017). https://doi.org/10.1007/978-3-319-68034-7_9
11. Rodríguez-Valdés, O., Vos, T.E.J., Aho, P., Marín, B.: 30 years of automated gui testing: A bibliometric analysis. In: Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R. (eds.) Quality of Information and Communications Technology. pp. 473–488. Springer International Publishing, Cham (2021)
12. Rosaria, S., Robinson, H.: Applying models in your testing process. Information and Software technology **42**(12), 815–824 (2000)
13. Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Elsevier (2010)