Preliminary Experiences in Requirements-based Security Testing

João Miranda[1][0000−0003−0976−3760], Ana C. R. Paiva[1,2][0000−0003−3431−8060], and Alberto Rodrigues da Silva[3,4][0000−0002−7900−9846]

[1] Faculty of Engineering, University of Porto, Portugal
[2] INESC TEC, Portugal
[3] INESC-ID, Portugal
[4] Instituto Superior Técnico, Universidade de Lisboa, Portugal
{up201802166,apaiva}@fe.up.pt, alberto.silva@tecnico.ulisboa.pt

Abstract. Software requirements engineers and testers generally define technical documents in natural languages, but this practice can lead to inconsistencies between the documentation and the consequent system implementation. Previous research has shown that writing requirements and tests in a structured way, with controlled natural languages like RSL, can help mitigate these problems. This study goes further, discussing new experiments carried out to validate that RSL (with its complementary tools, called "ITLingo Studio") can be applied in different systems and technologies, namely the possibility of applying the approach to integrate test automation capabilities in security testing. The preliminary conclusion indicates that, by combining tools such as ITLingo Studio and the Robot Framework, it is possible to integrate requirements and test specifications with test automation, and that would bring benefits in the testing process' productivity.

Keywords: Tests Automation · Security Testing · Test Case Generation · Requirements Engineering · RSL

1 Introduction

The vast majority of requirement specifications are written in natural languages. However, the use of natural language may lead to problems, such as ambiguity, incompleteness, inconsistency, and incorrectness [3] in the documentation, which, in turn, can lead to implementation errors. In addition, to define acceptance tests, testers rely on reading these documents. If the documents are not clear and complete, the test cases will also experience the same problems and the final system will be different from what is intended.

One of the approaches to mitigate these issues is by using controlled natural languages, like the ITLingo RSL (Requirements Specification Language) [16, 14], which provides a structured way of writing requirements and test cases [16]. RSL provides the capability to specify use cases, use case tests, and many other constructs in a common language.

Recently, Maciel et al. discussed an approach [6, 10] that aims to integrate and combine requirements and test cases specification with test automation tools, to improve the process of defining and executing acceptance tests. In particular, they discussed how to integrate ITLingo Studio (i.e., a Eclipse-based IDE that supports RSL and other languages) with the Robot Framework [14]. However, the software application and the examples they used to illustrate the approach were very simple. Thus, the goal of this work is to assess the applicability of the approach in different settings. First, we aim to assess the applicability on web applications developed using other technologies, namely based on the SPA (i.e., single page application) architecture. Second, we aim to preliminary check if it is possible to extend the approach to perform security testing.

This paper is structured in 5 sections. Section 2 presents the proposed testing process. Section 3 describes the experiments performed and discusses the results achieved. Section 4 refers and discusses the related work. Finally, section 5 presents final conclusion and open issues.

2 Proposed Testing Process

The proposed testing process intends to combine (1) both requirement and test specifications, defined in a tool like ITLingo Studio, with (2) test automation, supported by a tool like Robot Framework. This section briefly introduces the tools considered in this research, and overviews the proposed process.

2.1 Suported Tools: ITLingo Studio and Robot Framework

ITLingo-Studio is an Eclipse-based tool (i.e. a desktop IDE) for authoring IT technical documentation, such as requirements and tests (with the RSL language), project management plans (with the PSL language), or platform-independent application specifications (with the ASL). In the scope of this paper, we only consider the ITLingo RSL (or just RSL for brevity). RSL is a controlled natural language that helps the production of requirements and tests specifications in a systematic, rigorous and consistent way [3, 14, 16, 17]. RSL includes a rich set of constructs like use cases, goals, user stories, but also use case tests, data entities, actors, stakeholders, and many others (further details in [15, 10]. The ITLingo languages have been implemented with the Xtext framework (https://eclipse.org/Xtext/), so its specifications are rigorous and can be automatically validated and transformed into other representations and formats. Figure 1 depicts the ITLingo Studio with a test case specified in both RSL and Robot Framework languages.

Robot Framework (RF for short) is a generic open source automation framework that can be used for test automation and robotic process automation [11]. RF has an open and extensible architecture and can be integrated with other tools to create flexible automation solutions. Robot Framework provides a textual language with a keyword-based easy syntax, both machine and human-readable. Its capabilities can be extended by libraries implemented with Python or Java.

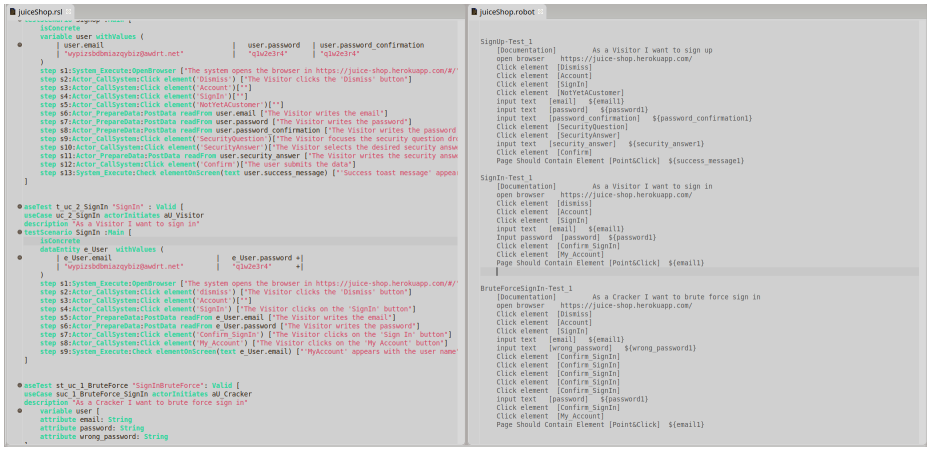

Fig. 1. RSL specification and test script in Robot.

In the scope of our research, the test cases defined in RSL [15] are generated to RF test scripts, and then these test scripts are executed by the RF's engine [12]. A key advantage of RF is its high modularity and extensibility, as it is platform-agnostic and thanks to its driver plugin architecture, the core framework does not require any previous knowledge of the system under test [13].

2.2 The Process

Figure 2 shows the approach originally proposed in [6], which considers an "end-to-end integration of requirements, test cases, and test scripts", namely by combining tools like the ITLingo Studio (with RSL specifications) and Robot Framework (with keyword-based test scripts).

This approach consists in a sequence of tasks, that varies from the specification of requirements until the execution of tests, as follows.

Requirements Specification. In the task "(1) Specify Requirements (manual)" the requirements are elicited and documented by requirement engineers using the RSL language. This may also involve other stakeholders (like testers or domain experts) for validation purposes. This task favors with a systematic specification of requirements and can be done manually with the ITLingo Studio (that is an Eclipse IDE for authoring RSL specs) or an Excel template.

Test cases specification and validation. Task "(2) Specify Test Cases (semi-auto)" uses the requirements defined in the previous step to generate test cases, namely considering the mappings between use case and use case test, as defined in [16]. Task "(3) Refine Test Cases (manual)" allows the tests engineers to refine and complete the (previously generated) test cases with all the relevant test scenarios, steps, etc. Also, the tests generated from the previous task may be subject to manual validation and this could result in new test cases which are

then added to the final specification. This test case validation is also a human-intensive and time-consuming task.

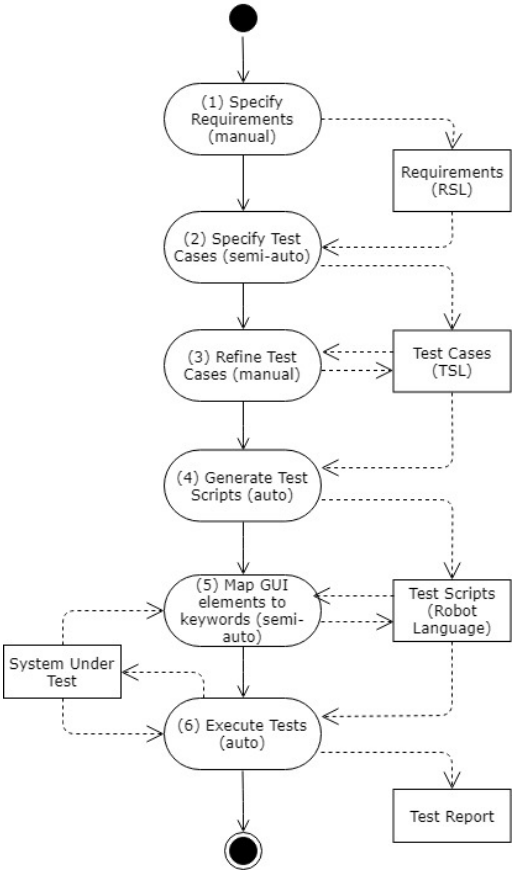

Fig. 2. Proposed testing approach, as originally defined in [6].

Test scripts generation. Once the specification and validation of test cases are completed, the task "(4) Generate Test Scripts (auto)" consists in the generation of test scripts, which can be then executed by a test automation tool, like the Robot Framework. This generation is based on the mappings established between the RSL specification and the concrete syntax of the Robot Framework [6].

Map the UI elements. The user interface (UI) of a web application involves several UI elements [6]. To properly support the test automation based on the acceptance tests, these generic UI elements have to be mapped to concrete UI variables defined in the test scripts so as to act over related UI elements during test case execution. The establishment of this map is the purpose of task "(5) Map GUI elements to keywords (semi-auto)". However, using a web scrapper browser

extension, as the Selenium WebDriver (https://www.selenium.dev/projects/), it is possible to automatically capture and generate scripts that successfully map these UI elements with the test scripts. Unfortunately, most of these cases depend on the proper keywords mapping, which is hard to be fully automated.

Execute test scripts. Task "(6) Execute tests (auto)" takes into consideration the test scripts (previously generated, and with their UI variables mapped to concrete UI elements) and execute them on the system under test (SUT). This test automation task produces an artifact "Test Report", which includes the log details of its execution.

3 Security Testing based on RSL

This section describes two sets of experiments conducted to explore different objectives. The first one aims to apply the RSL testing approach, as it is, but with a web application with different characteristics than the one considered originally in [6]. There are differences in technologies used to develop the Automation Practice website (used in the previous study) and OWASP Juice Shop. While Automation Practice is developed in PHP [6, 13], OWASP Juice Shop is developed using Angular and Angular-Material library for styling [8]. The usage of Angular, in many cases, prevented the correct mapping of the UI elements to their respective keywords. This happens mainly because: there are unspecified element identifiers, elements are too generic, are not standard, or have overlays. These issues prevented the correct map of inputs, buttons and dropdown buttons, which are not found at all. The second set of experiments aims to evaluate if the RSL-based testing approach can be used as well to test security aspects.

The application used in these experiments is the Juice Shop web application, which is an open-source web application, entirely written in JavaScript, and listed in the OWASP VWA (Vulnerable Web Applications) directory [8]. Juice Shop includes a large number of hacking challenges and vulnerabilities that a user is supposed to exploit. As suggested in Figure 3, Juice Shop mimics an e-commerce website that sells juices and vegetables, and so, it is a good application example for acceptance and security testing.

3.1 First Experiments

The first set of experiments aims to check the applicability of the RSL testing approach. The first step of these experiments was to define the requirements specification. For that we defined the data entities (e.g. user, product, and others), actors (e.g., visitor and cracker), use cases (e.g., sign-up, sign-in), and respective test cases.

The second step was to define test cases to check the behavior of the application related to the authentication functionality, namely: (1) Sign up; (2) Sign in (Spec. 1.1); and (3) Sign in attempt by simulating a brute force attack (Spec. 1.3).

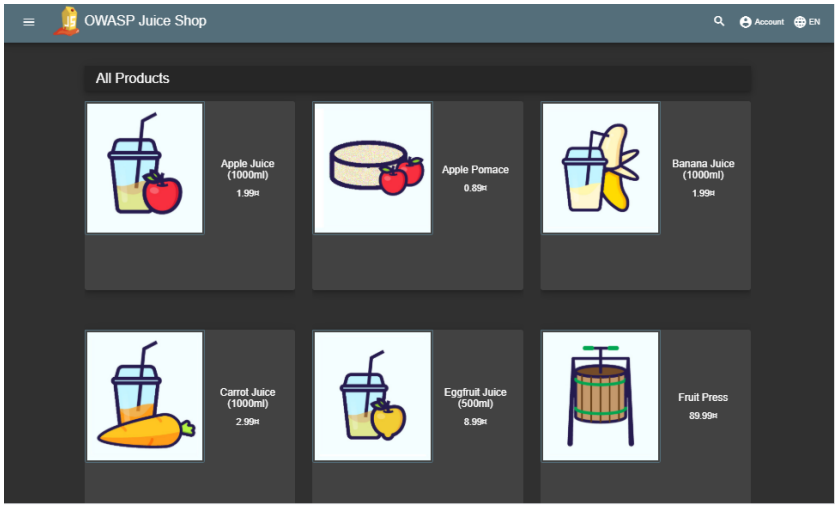

Fig. 3. Screenshot of the Juice Shop Application.

The test cases consist of a user trying to sign-up and then to sign-in. Afterwards, the goal is a brute force attempt by deliberately sending wrong passwords and, in the last time, submitting the correct password.

After the specification of these test cases, the RSL engine generates automatically the respective test scripts defined as Robot Framework (".robot") files.

Then, to support establishing the mapping between keywords and UI elements of the GUI, we used two tools: the one developed in previous work [6] and web scraper techniques to select elements to be used by the Robot Framework.

Web scraping can be achieved by mapping specific elements of a web page through their XML path (X-Path) or CSS selectors, generally speaking, X-Path is more trustworthy as it is possible to achieve certain elements selection more precisely [19, 5]. The X-Path of a given element can be constructed by its XML absolute path or through additional selectors, that allow to "take shortcuts" on a document's structure in order to reach some element [19, 5].

The solution we adopted to overcome this technical problem was to resort to X-Path mapping instead of CSS mapping. However, a X-Path based approach has also some issues, namely: most of the web-scrapping tools that support X-Path are not free; the free web-scrapping extension that supported X-Path was not accurate leading to unnecessary rework, the free web-scrapping extension that was used by default tries to find the X-Path using special selectors which may include CSS, classes, and relative selectors, thus we opted to use always the absolute X-Paths.

Sign-Up Test Case

The idea of the "Sign-Up" is to try to test an user registration by informing the correct parameters and check if the user is successfully created. This test can be

extended to verify if the web application allows the same user to be registered twice, etc.

The test results were successful although some mapping issues with custom elements of angular material such as the select or the input were detected and fixed by the usage of X-Path.

Sign-In Test Case

The idea of the "Sign-In" test is to check if a previously registered user can sign in according to the documentation by providing the e-mail and their respective password. The result of this test is a success even though it presented the same issues as the previous test. A successful login attempt is made and the user is correctly authenticated.

Spec. 1.1 shows the Sign-In test case specified in the RSL language. Spec. 1.2 shows the equivalent specification in the Robot Framework language.

```
UseCaseTest t_uc_2_SignIn "Sign-In test case" : Valid [
    useCase uc_2_SignIn actorInitiates aU_Visitor
    description "As a Visitor I want to sign in"
    testScenario SignIn :Main [
         isConcrete
         dataEntity e_User  withValues (
          | e_User.email         | e_User.password  +|
          | "user001@awdrt.net" | "q1w2e3r4"       +|
          )
        step s1:System_Execute:OpenBrowser ["The system opens the browser in
            https://juice-shop.herokuapp.com/#/"]
        step s2:Actor_CallSystem:
            Click element('dismiss')
            ["The Visitor clicks the 'Dismiss' button"]
        step s3:Actor_CallSystem:
            Click element('Account')[""]
        step s4:Actor_CallSystem:
            Click element('SignIn')
            ["The Visitor clicks on the 'SignIn' button"]
         step s5:Actor_PrepareData:
             PostData readFrom e_User.email
             ["The Visitor writes the email"]
         step s6:Actor_PrepareData:
             PostData readFrom e_User.password
             ["The Visitor writes the password"]
         step s7:Actor_CallSystem:
             Click element('Confirm_SignIn')
             ["The Visitor clicks on the 'Sign In' button"]
         step s8:Actor_CallSystem:
             Click element('My_Account')
             ["The Visitor clicks on the 'My Account' button"]
         step s9:System_Execute:
             Check elementOnScreen(text e_User.firstName)
             ["'MyAccount' appears with the user name"]
    ]
]
```

Spec. 1.1. User Sign In test case (RSL spec).

```
*** Settings ***
Documentation               This is a basic test
Library                     Selenium2Library
*** Variables ***

${email1}    wypizsbdbmiazqybiz@awdrt.net
${password1}    q1w2e3r4

*** Test Cases ***

SignIn-Test_1
  [Documentation] As a Visitor I want to sign in
  open browser  https://juice-shop.herokuapp.com/
  Click element
    //button[contains(@class,'mat-focus-indicator mat-raised-button')]
    /following-sibling::button[1]
  Click element
    (//mat-toolbar-row[@class='mat-toolbar-row']//button)[3]
  Click element
    navbarLoginButton
```

```
input text
  (//input[contains(@class,'mat-input-element mat-form-field-autofill-control')])[2]
  ${email1}
Input password
  (//input[contains(@class,'mat-input-element mat-form-field-autofill-control')])[3]
    ${password1}
Click element
  (//div[@id='login-form']//button)[2]
Click element
  (//button[contains(@class,'mat-focus-indicator mat-menu-trigger')])[1]
Page Should Contain Element
  (//simple-snack-bar[@class='mat-simple-snackbar ng-star-inserted']//span)[1]
    ${email1}
```

Spec. 1.2. User Sign In test (Robot Framework spec).

Multiple Invalid Login Attempts Test Case

This test aims to try and check, by repetition, if there is any kind of protection against multiple incorrect login attempts such as IP blocking, account locking, etc. This attempt tries emulate the incorrect submission of a username/password combination before attempting to login with the correct password to check if the user/account was blocked or not. The repeated submission of requests to a determined application could be also identified as a potential Denial of Service attack and an application that has protection against such attacks could also block a malicious user attempt after a number of login attempts.

The main point of this test is to check if after 3 to 5 attempts of requests with invalid passwords the system will present any kind of lock or captcha to block malicious attempts to sign-in. This test results showed that the web application has no countermeasures against brute force attacks as after 5 attempts it was possible to keep retrying until the password was correct.

```
UseCaseTest st_uc_1_BruteForce "SignInBruteForce": Valid [
    useCase suc_1_BruteForce_SignIn actorInitiates aU_Cracker
    description "As a Cracker I want to brute force sign in"
        variable user [
        attribute email: String
        attribute password: String
        attribute wrong_password: String  ]
    testScenario BruteForceSignIn : Main [
         isConcrete
         variable user  withValues (
            | user.email          | user.wrong_password      | user.password      +|
            | "user001@awdrt.net"| "q1w2e3r4t5"             | "q1w2e3r4"         +|
        )
        step s1:System_Execute:
            OpenBrowser ["The system opens the browser
            in https://juice-shop.herokuapp.com/#/"]
        step s2:Actor_CallSystem:
            Click element('Dismiss')
            ["The Visitor clicks the 'Dismiss' button"]
        step s3:Actor_CallSystem:
            Click element('Account')[""]
        step s4:Actor_CallSystem:
            Click element('SignIn')
            ["The Visitor clicks on the 'SignIn' button"]
        step s5:Actor_PrepareData:
            PostData readFrom user.reg_email
            ["The Visitor writes the email"]
        step s6:Actor_PrepareData:
            PostData readFrom user.reg_wrong_password
            ["The Visitor writes the wrong password"]
        step s7:Actor_CallSystem:
            Click element('Confirm_SignIn')
            ["The Visitor clicks on the 'Sign In' button"]
        step s8:Actor_CallSystem:
            Click element('Confirm_SignIn')
            ["The Visitor clicks on the 'Sign In' button"]
        step s9:Actor_CallSystem:
            Click element('Confirm_SignIn')
            ["The Visitor clicks on the 'Sign In' button"]
        step s10:Actor_CallSystem:
            Click element('Confirm_SignIn')
            ["The Visitor clicks on the 'Sign In' button"]
```

```
        step s11:Actor_CallSystem:
            Click element('Confirm_SignIn')
            ["The Visitor clicks on the 'Sign In' button"]
        step s12:Actor_PrepareData:
            PostData readFrom user.reg_password
            ["The Visitor writes the correct password"]
        step s13:Actor_CallSystem:
            Click element('Confirm_SignIn')
            ["The Visitor clicks on the 'Sign In' button"]
        step s14:Actor_CallSystem:Click element('My_Account')
             ["The Visitor clicks on the 'My Account' button"]
    step s15:System_Execute:Check elementOnScreen(text e_User.firstName)
         ["'MyAccount' appears with the user name"]
     ]
]
```

Spec. 1.3. Multiple invalid login attempts test case (RSL spec).

The result of this first set of tests showed that, although it is possible to perform such validations using RSL and ITLingo Studio, this may be a very repetitive and time consuming task, as the language currently does not support iteration operators, thus requiring manual implementation of each invalid sign in attempt. In the case of the Juice Shop application, after 5 attempts with wrong passwords, the test login with a correct password and was successfully authenticated, and thus, this shows that the Juice Shop application is not protected against brute force sign-in attacks.

3.2 Second Experiments

The goal of the second set of experiments is to check if it is possible to apply the proposed approach in security testing.

We analyzed the documentation of OWASP Juice Shop [8] to identify and understand the involved exploits. Most of its exploits referred to the following vulnerabilities: SQL Injection; XSS; Vulnerable Components; Sensitive Data Exposure; Broken Authentication; Broken Access Control.

Many of these exploits require tools that have to be combined with ITLingo Studio, namely to include features such as: use browser developer mode, use external programs to inspect requests, and penetration testing tools. In this preliminary study, we just explored the following vulnerabilities: SQL Injections and XSS based on UI inputs.

SQL Injection attack - Abuse Login as Administrator
The goal of this test is to check if it is possible to execute an SQL injection attack on the system. This requires some previous knowledge of the system architecture, but in general, this knowledge is available.

Spec. 1.4 shows an RSL specification of the "Abuse Login as Administrator" test case. This test is a simple example of an SQL injection attack. The goal of this test is to break the login query by sending a special sequence of characters (i.e., the string "' OR 1=1–") that returns true, and that allows to login with the first user of the application's database: that happens to be the "administrator" user! With this test, it was possible to detect an important system vulnerability, since it is possible to incorrectly login as an administrator and, thus, login with higher privileges accessing information that should not be supposed to.

This test resulted in a successful login as administrator, which in a real scenario would allow an attacker to execute actions that normally should not be available to regular users.

```
UseCaseTest st_uc_1_SQLInject "" : Invalid [
    useCase suc_2_SQLi_Admin_User actorInitiates aU_Cracker
    description "As a Cracker I want to login as
        Administrator using SQLi"
    variable query [
        attribute login: String
        attribute password: String
    ]
    testScenario SQLinjectionAdministratorSignIn : Main [
        isConcrete
        variable query withValues (
        |query.login     |query.password+|
        |"' OR 1=1--"    |""             +|
    )
          step s1:System_Execute:OpenBrowser
            ["The system opens the browser in
            https://juice-shop.herokuapp.com/#/"]
        step s2:Actor_CallSystem:
            Click element('Dismiss')
            ["The Visitor clicks the 'Dismiss' button"]
        step s3:Actor_CallSystem:
            Click element('Account')[""]
        step s4:Actor_CallSystem:
            Click element('SignIn')
            ["The Visitor clicks on the 'SignIn' button"]
        step s5:Actor_PrepareData:
             PostData readFrom query.login
             ["The Cracker writes the query"]
        step s6:Actor_PrepareData:
             PostData readFrom query.password
             ["The Cracker writes random data"]
        step s7:Actor_CallSystem:
             Click element('Confirm_SignIn')
             ["The Visitor clicks on the 'Sign In' button"]
        step s8:Actor_CallSystem:
             Click element('My_Account')
             ["The Visitor clicks on the 'My Account' button"]
        step s9:System_Execute:
            Check elementOnScreen(text query.login)
            ["'MyAccount' appears with the user name"]
    ]
```

Spec. 1.4. RSL Test Steps for SQL injection.

Embed IFrame which calls Another Website on Screen
This test consists in creating an address that would return a specific set of characters, then by inputting an IFrame that references that address and submit the search query to test if the system renders the input data as HTML.

This test aims to check if it is possible to fill in an input field with an IFrame redirecting to a new location that does not belong to the application under test. Indeed, it was possible to confirm this vulnerability since after the search input form, the system redirected to the new location. However, this test required the local execution of the OWASP Juice Shop as it is needed to access a local API.

With this preliminary study it is possible to conclude that the proposed approach (with tools like ITLingo Studio and Robot Framework) can be adopted to test security vulnerabilities. Nevertheless, it need to be extended and combined with additional functionalities and other cyber-security specific tools.

4 Related Work

The greatest problem of using natural language for documenting requirements is that it usually leads to well-known problems, such as ambiguity, incompleteness,

inconsistency, and incorrectness [3]. Ambiguity is a particularly serious problem, as it allows for multiple interpretations of the same requirement [3]. This issue can make the automation of some subsequent activities of the software development process more difficult or even unfeasible. One way to overcome the ambiguity problem is to use structured languages and/or models.

4.1 Test Cases Generation

One of the less treated phases during a software development project is the testing and validation phase [4]. Therefore, a good way to combat this trend would be to increase the automation of testing activities, with the aim of improving the final quality of a software development project.

One test activity that is a candidate for automation is the generation of test cases. It is common to find research works that generate test cases from models of the system under development. The models adopted and data used by different approaches may vary significantly [7, 2, 18]. Besides this, it is also possible to find some works that try to generate test cases from requirements specified in natural languages [1].

Requirements specification in natural language are common. However, generating automatically test cases from them is challenging because they are often written in a non-standardized way.

The work presented in [1] uses Natural Language processing tools to generate automatically test cases from functional requirements written in natural language. The process followed is based on three steps, namely: (1) Functional Requirements specification; (2) Applying Natural Language Processing technique to analyze requirements and extract the data necessary for the construction of test cases; and (3) Generate the test cases from the extracted data.

To enable the generation of test cases, the requirements' document must be written according to a certain level of standardization, otherwise no results are produced.

An approach based on models is presented in [4] where test cases are generated from a *Round-Strip Strategy* and *Extended Use Cases*. In this work, the functional requirement may be treated as a graph or state machine which allows to apply path-finding algorithms. Each path can be used as a test case. In complement, the authors use a Category-Partition method on Use Cases to identify subsets of the domain for executing concrete pieces of the behavior.

In [9], the authors generate test cases from user execution traces, i.e., all user interactions with the web application are saved so as to work as regression test cases when the application suffers updates in the context of its maintenance. The original traces are extended by applying mutation operators that were designed to generate different sequences of interaction and ultimately exercise more behavior of the web application under test.

One of the challenges that these approaches have to deal with is to identify the User Interface (UI) elements to act upon when executing the final generated test scripts over the application under test. Sometimes this mapping between concepts of the specification level (either model elements or textual keywords)

and UI elements is established manually which may require some effort. Although the keyword mapping is still done manually in ITLingo Studio, it does not require language processing capabilities since the specification is structured and it is simple to get all keywords that must be mapped to UI elements for test script execution.

5 Conclusion

At the moment, RSL and ITLingo Studio have some dependencies and work well in some specific scenarios. In the future, this approach should be extended and improved to provide a more simplified user experience.

One aspect that should be considered for improvement is to allow working with more types of applications based on different technologies. In addition, changing from the exclusive CSS mapping to the generic X-Path mapping will allow the use of CSS and X-Path mapping in the selection of UI elements. This change combined with a browser extension capable of inferring the absolute X-Path and, if necessary, in a second step, offering CSS alternatives, may allow a more direct way of mapping the elements of the user interface to the desired keywords.

Regarding the language, we noticed that the entire approach would benefit from some RSL extensions. In particular, it should be considered extending the RSL language with the ability to make direct references to entities. This may be useful for defining test scenarios. Another aspect is to extend RSL with a mechanism supporting iterations inside test scenarios. This problem was noticed when defining the "brute force" test case in Spec 1.3.

Regarding the overall approach, we think that it would benefit from adding a test data generation technique. This would ease the definition of the test cases. In addition, ITLingo's usability would improve if the components were integrated in a single environment. As it is now, it requires the use of different technologies and different IDEs to successfully run all necessary scripts.

Finally, and related to security tests, the tool has some limitations to perform this type of tests. However, some of the mentioned problems may be overcome if implemented in the scripting engine.

References

1. Ansari, A., Shagufta, M.B., Sadaf Fatima, A., Tehreem, S.: Constructing Test cases using Natural Language Processing. Proceedings of the 3rd IEEE International Conference on Advances in Electrical and Electronics, Information, Communication and Bio-Informatics, AEEICB (2017). https://doi.org/10.1109/AEEICB.2017.7972390
2. Barbosa, A., Paiva, A.C.R., Campos, J.C.: Test case generation from mutated task models. In: Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011. ACM (2011). https://doi.org/10.1145/1996461.1996516

3. de Almeida Ferreira, D., da Silva, A.R.: RSLingo: An information extraction approach toward formal requirements specifications. In 2nd IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE (2012). https://doi.org/10.1109/MoDRE.2012.6360073
4. Gutiérrez, J., Aragón, G., Mejías, M., Domínguez Mayo, F.J., Ruiz Cutilla, C.M.: Automatic Test Case Generation from Functional Requirements in NDT. In: Current Trends in Web Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
5. Jr, V.S.: An Introduction to XPath: How to Get Started. `https://blog.scrapinghub.com/2016/10/27/an-introduction-to-xpath-with-examples` (2016)
6. Maciel, D., Paiva, A.C., Da Silva, A.R.: From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach. ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (2019). https://doi.org/10.5220/0007679202650272
7. Moreira, R.M.L.M., Paiva, A.C.R., Nabuco, M., Memon, A.: Pattern-based GUI testing: Bridging the gap between design and quality assurance. Softw. Test. Verification Reliab. **27**(3) (2017). https://doi.org/10.1002/stvr.1629
8. OWASP: OWASP Juice Shop - demo and testing instance. `https://juice-shop.herokuapp.com/#/`
9. Paiva, A., Restivo, A., Almeida, S.: Test case generation based on mutations over user execution traces. Software Quality Journal (05 2020). https://doi.org/10.1007/s11219-020-09503-4
10. Paiva, A.C.R., Maciel, D., da Silva, A.R.: From Requirements to Automated Acceptance Tests with the RSL Language. In: Evaluation of Novel Approaches to Software Engineering. Springer International Publishing, Cham (2020)
11. Robot-Framework-Foundation: Robot Framework. `https://robotframework.org/`
12. Rwemalika, R., Kintis, M., Papadakis, M., Le Traon, Y., Lorrach, P.: On the evolution of keyword-driven test suites. Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST (2019). https://doi.org/10.1109/ICST.2019.00040
13. Selenium: Automation Practice. `http://automationpractice.com/index.php?id{\_}cms=4{\&}controller=cms`
14. da Silva, A.R.: Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-Level Language. Proceedings of the 22nd European Conference on Pattern Languages of Programs (2017)
15. da Silva, A.R.: Rigorous Specification of Use Cases with the RSL Language. In: 28th International Conference on Information Systems Development - IDS (08 2019)
16. da Silva, A.R., Paiva, A.C.R., da Silva, V.E.R.: A Test Specification Language for Information Systems Based on Data Entities, Use Cases and State Machines. Communications in Computer and Information Science, vol. 991. Springer (2018). https://doi.org/10.1007/978-3-030-11030-7_20
17. da Silva., A.R., Paiva, A.C.R., da Silva., V.E.R.: Towards a Test Specification Language for Information Systems: Focus on Data Entity and State Machine Tests. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - MODELSWARD. INSTICC, SciTePress (2018). https://doi.org/10.5220/0006608002130224

18. Silva, P., Paiva, A.C.R., Restivo, A., Garcia, J.E.: Automatic Test Case Generation from Usage Information. In: 11th International Conference on the Quality of Information and Communications Technology, QUATIC. IEEE Computer Society (2018). https://doi.org/10.1109/QUATIC.2018.00047
19. W3School: XML and XPath. `https://www.w3schools.com/xml/xml{\_}xpath.asp`