From Requirements to Automated Acceptance Tests of Interactive Apps: An Integrated Model-based Testing Approach

Daniel Maciel[1], Ana C. R. Paiva[1] and Alberto Rodrigues da Silva[2]
[1]*INESC TEC, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal*
[2]*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal*

Keywords: Requirements Specification Language (RSL), Test Case Specification, Model-based Testing (MBT), Test Case Generation, Test Case Execution.

Abstract: Frequently software testing tends to be neglected at the beginning of the projects, only performed on the late stage. However, it is possible to benefit from combining requirement with testing specification activities. On one hand, acceptance tests specification will require less manual effort since they are defined or generated automatically from the requirements specification. On the other hand, the requirements specification itself will end up having higher quality due to the use of a more structured language, reducing typical problems such as ambiguity, inconsistency and incorrectness. This research proposes an approach that promotes the practice of tests specification since the very beginning of projects, and its integration with the requirements specification itself. It is a model-driven approach that contributes to maintain the requirements and tests alignment, namely between requirements, test cases, and low-level automated test scripts. To show the applicability of the approach, two complementary languages are adopted: the ITLingo RSL that is particularly designed to support both requirements and tests specification; and the Robot language, which is a low-level keyword-based language for the specification of test scripts. The approach includes model-to-model transformation techniques, such as test cases into test scripts transformations. In addition, these test scripts are executed by the Robot test automation framework.

1 INTRODUCTION

Software systems are constantly evolving becoming more complex, which increases the need for efficient and regular testing activity to ensure quality and increase the product confidence. Software systems' quality is usually evaluated by the software product's ability to meet the implicit and explicit customer needs. For this purpose, it is important that customers and developers achieve a mutual understanding of the features of the software that will be developed.

Requirements Engineering (RE) intends to provide a shared vision and understanding of systems among the involved stakeholders and throughout its life-cycle. The system requirements specification (SRS) is an important document that helps to structure the system's concerns from an RE perspective and offers several benefits, already reported in literature (Cockburn, 2001; Kovitz, 1998; Robertson and Robertson, 2006; Withall, 2007), such as the establishment of an agreement between users and developers, support validation and verification of the project scope, and support future system maintenance activities. The problem is that the manual effort required to produce requirements specifications is high and it suffers from problems, such as, incorrectness, inconsistency, incompleteness, and ambiguity (Kovitz, 1998; Robertson and Robertson, 2006; Pohl, 2010).

ITLingo is a long term initiative with the goal to research, develop and apply rigorous specification languages in the IT domain, namely Requirements Engineering, Testing Engineering and Project Management (Silva, 2018). ITLingo adopts a linguistic approach to improve the rigorous of technical documentation (e.g., SRS, test case specification, project plans) and, as a consequence, to promote productivity through re-usability and model transformations, as well as promote quality through semi-automatic validation techniques.

RSL (Requirements Specification Language) is a controlled natural language integrated in ITLingo which helps the production of requirements specifications in a systematic, rigorous and consistent way (da Silva, 2017). RSL includes a rich set of constructs logically arranged into views according to RE-

Maciel, D., Paiva, A. and Rodrigues da Silva, A.
From Requirements to Automated Acceptance Tests of Interactive Apps: An Integrated Model-based Testing Approach.
DOI: 10.5220/0007679202650272
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 265-272
ISBN: 978-989-758-375-9

specific concerns that exist at different abstraction levels, such as business, application, software or even hardware levels.

Software testing can be used to track and evaluate the software development process by measuring the number of tests that pass or fail and by performing continuous regression testing that allows to maintain the product quality alerting developers of possible defects as soon as the code is changed. Among different types of tests, acceptance tests (Christie, 2008) are those that have a greater relationship with the requirements. They are used to test with respect to the user needs, requirements and business processes, and are conducted to determine whether or not a system satisfies the acceptance criteria and to enable users, customers or other authorized entities to determine whether or not shall accept the system (ISTQB, 2015). In order to improve the acceptance testing and requirements specification activities, it may be advantageous to perform these activities in parallel which, in addition to increasing the quality of the requirements, also allows to reduce the time and resources spent with them. Even though, starting the testing activities at the beginning of the project when the requirements are elicited is considered a good practice, this is not always the case because requirements elicitation and testing are separate in traditional development processes.

This research proposes and discusses an approach based on the Model-based Testing (MBT) technique (ISTQB, 2015) capable of promoting the initiation of testing activities earlier when specifying requirements. MBT is a software testing approach that generates test cases from abstract representations of the system, named models, either graphical (e.g., Workflow models (Boucher and Mussbacher, 2017), PBGT (Moreira et al., 2017; Moreira and Paiva, 2014)) or textual (e.g., requirements documents in an intermediate format)(Paiva, 2007).

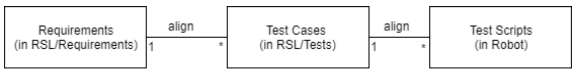


Figure 1: Approach terminologies.

Following the Figure 1, this approach uses RSL *Requirements* specifications produced through a set of constructs provided by the language according to different concerns. Then, each *Requirement* is aligned with RSL *Test Cases* specifications. From the RSL *Test Cases* specifications, it is possible to align and generate *Test scripts* that can be executed automatically by the Robot[1] test automation tool over the System Under Test (SUT).

[1]http://robotframework.org/

This paper is organized in 6 sections. Section 2, overviews the RSL language, showing its architecture, levels of abstraction and concerns. Section 3 introduces the concepts of the selected test automation tool, the Robot Framework. Section 4 presents the proposal approach with a running and illustrative example. Section 5 identifies and analyzes related work. Finally, Section 6 presents the conclusion and briefly mention the future work.

2 RSL LANGUAGE

ITLingo research initiative intends to develop and apply rigorous specification languages for the IT domain, such as requirements engineering and testing engineering, with the RSL (Silva et al., 2018). RSL provides a comprehensive set of constructs that might be logically arranged into views according to specific concerns. These constructs are defined by *linguistic patterns* and represented textually according to concrete *linguistic styles*. RSL is a process- and tool-independent language, i.e., it can be used and adapted by different organizations with different processes or methodologies and supported by multiple types of software tools (Silva, 2018). This paper focuses on the RSL constructs particularly supportive of use case approaches (e.g. actors, data entities and involved relationships).

RSL constructs are logically classified according to two perspectives (Silva, 2018): abstraction level and specific concerns they address. The abstraction levels are: business, application, software and hardware levels. On the other hand, the concerns are: active structure (subjects), behaviour (actions), passive structure (objects), requirements, tests, other concerns, relations and sets. (This paper focuses the discussion on the requirements and tests concerns.)

2.1 Requirements Specification

Figure 2 shows a partial view of the RSL metamodel that defines a hierarchy of requirement types, namely: goals, functional requirement, constraint, use case, user story and quality requirement. (This paper focuses the discussion on only the *UseCase* requirement type.)

RSL specifications based on *Use Cases* can involve the definition of some views with the respective constructs and inherent relations:

- *DataEntity View:* defines the structural entities that exist in an information system, commonly associated to data concepts captured and identified from the domain analysis. A *Data Entity* denotes

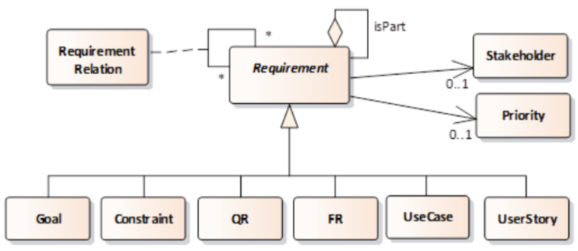


Figure 2: RSL partial metamodel: The hierarchy of requirements.

an individual structural entity that might include the specification of attributes, foreign keys and other checkdata constraints;

- *DataEntityCluster View:* denotes a cluster of several structural entities that present a logical arrangements among themselves;
- *Actor view*: defines the participants of *Use Cases* or *user stories*. Represent end-users and external systems that interact directly with the system under study, and in some particular situations can represent timers that trigger the start of some *Use Cases*;
- *Use Case View* defines the *use cases* of a system under study. Traditionally a use case means a sequence of actions that one or more actors perform in a system to obtain a particular result (Jacobson and et al, 2015);

2.2 Tests Specification

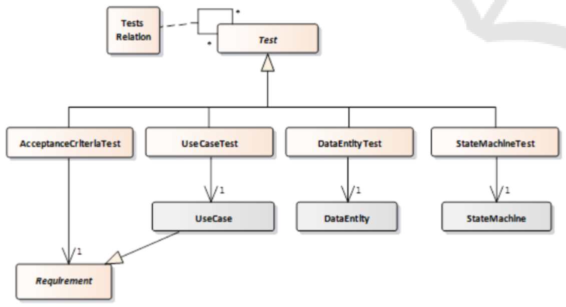


Figure 3: RSL partial metamodel: the hierarchy of Tests.

RSL supports the specification and generation of software tests, directly from requirements specifications. As showed in Figure 3, RSL provides a hierarchy of Test constructs and supports the specification of the following specializations of test cases (Silva, 2018):

- *DataEntityTest* apply equivalence class partitioning and boundary value analysis techniques over the domains defined for the DataEntities (Bhat and Quadri, 2015) in RSL *DataEntities*;
- *UseCaseTest* explores multiple sequences of steps defined in RSL use cases' scenarios, and associates data values to the involved data entities;
- *StateMachineTest* apply different algorithms to traverse the RSL state machines so that it is possible to define different test cases that correspond to valid or invalid paths through the respective state machine;
- *AcceptanceCriteriaTest* define acceptance criteria based on two distinct approaches: scenario based (i.e., Given-When-Then pattern) or rule based; this test case is applied generically to any type of RSL Requirement

Regardless of these specializations, a *Test* shall be defined as *Valid* or *Invalid* depending on the intended situation. In addition, it may establish relationships with other test cases through the *TestsRelation*; these relationships can be further classified as *Requires*, *Supports*, *Obstructs*, *Conflicts*, *Identical*, and *Relates*.

3 ROBOT FRAMEWORK

Test cases can be executed manually by the tester or automatically by a test automation tool. When a test case is executed manually, the tester must perform all test cases, having to repeat the same tests several times throughout the product life cycle. On the other hand, when test cases are run automatically, there is the initial effort to develop test scripts, but from there, the execution process will be automatic. So if a test case has to run many times, the automation effort will be less than the effort of frequent manual execution.

The Robot framework stands out for its powerful keyword-based language that includes out-of-the-box libraries. Robot does not require any kind of implementation, since it is possible to use keywords with implicit implementations (with the use of specific libraries such as Selenium[2]). Robot is an open source framework, related to the acceptance test-driven development (ATDD) (ISTQB, 2014), which is independent of the operating system and is natively implemented in Python and Java, and can be executed in Jython (JVM) or IronPython (.NET) .

The script structure is simple and can be divided into four sections. The first section, *Settings*, where paths to auxiliary files and libraries used are configured. The second section, *Variables*, specifies the list of variables that are used, as well as the associated values. The third and most important section is the *Test Cases*, where test cases are defined. Lastly, the

[2]https://www.seleniumhq.org/

Keywords section define custom keywords to implement the test cases described in the Test Cases section. Among the sections mentioned above, only the *Test Cases* section is mandatory.

As can be seen in the example presented in Listing 1, the libraries used are initially defined. One of the most used is the Selenium library that introduces interactive applications test-related keywords, such as 'Open Browser' and 'Input text'. The variables section, assigns 'Blouse' to variable 'product' so, whenever 'product' is used, it will have value 'Blouse'. The Keywords section defines keywords and their parameters. In test cases using keywords, the values are assigned to the corresponding parameters by placing the values in the same place where parameters are defined.

Listing 1: Robot Framework specification example.

```
*** Settings ***
Documentation  Web Store Acceptance Test
Library    Selenium2Library

*** Variables ***
${product}   Blouse

*** Test Cases ***
Login
 Open the browser on <www.http://
     automationpractice.com>
 Input Text id=searchBar ${product}
 ...

*** Keywords ***
Open the browser on <$(url)>
   Open Browser $(url)
```

4 PROPOSED APPROACH

This research intends to encourage and support both requirements and tests practices, namely by generating test cases from requirements or at least foster the alignment of such test cases with requirements.

The proposed approach (as defined in Figure 4) begins with the (1) requirements specification that serves as a basis for the (2) test cases specification, which can be further (3) refined by the tester. Then, (4) tests scripts are generated automatically from the high-level test cases, and (5) associated the Graphical User Interface (GUI) elements. Finally, (6) these test scripts are executed generating a test report.

To illustrate and discuss the suitability of the approach, we applied it on an web application: the Web Store[3], because this was specifically designed to be

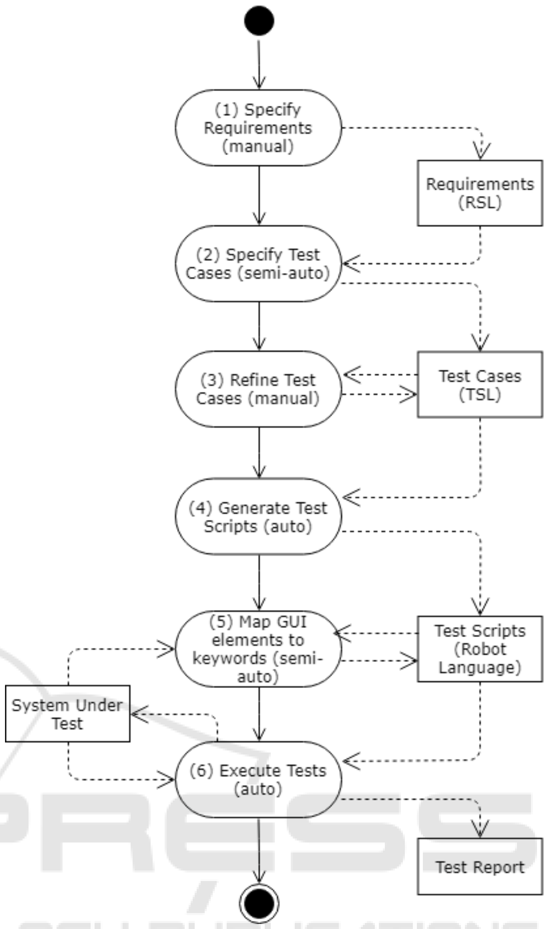

(1) Specify Requirements (manual)
Requirements (RSL)
(2) Specify Test Cases (semi-auto)
(3) Refine Test Cases (manual)
Test Cases (TSL)
(4) Generate Test Scripts (auto)
(5) Map GUI elements to keywords (semi-auto)
Test Scripts (Robot Language)
System Under Test
(6) Execute Tests (auto)
Test Report

Figure 4: Proposed approach (UML activity diagram).

tested.

4.1 Specify Requirements

The first task is the requirements definition that usually involves the intervention of requirements engineers, stakeholders and eventually testers. In this example, the specification focuses on the most relevant RSL constructors at the application and software level, namely: *Actor, UseCase, DataEntity*. Listing 2 shows the constructs defined for the "search of a product in the store".

4.2 Specify Test Cases

Use Case Tests are derived from the various process flows expressed by a RSL *UseCase*. Each test contains multiple test scenarios. A test scenario encompasses of a group of test steps and shall be executed by an actor (Silva et al., 2018). *UseCaseTest* construct begins by defining the test set, including *ID*, *name* and

[3]http://automationpractice.com

the use case *type*. Then it encompasses the references keys [Use Case] indicating the *Use Case* in which the test is proceeding and [DataEntity] referring to a possible data entity that is managed (Silva et al., 2018).

Listing 2: Example of a RSL specification of Data Entity Actor and UseCase.

```
DataEntity e_Product "Product" : Master [
 attribute ID "ID" : Integer [isUnique]
 attribute title "title" : Text [isNotNull]
 attribute price "Price" : Integer [
      isNotNull]
 attribute composition "Composition" : Text
 attribute style "Style" : Text
 attribute properties "Properties" : Text
 primaryKey (ID)]

Actor aU_Customer "Customer" : User

UseCase uc_Search "Search␣Products" :
    EntitiesSearch [
 actorInitiates aU_Customer
 dataEntity e_Product]
```

4.3 Refine Test Cases

The generated test cases may be subject to manual refinements (e.g., assign values to entities and create temporary variables), resulting in other test cases. The *DataEntities* and the temporary *Variables* are fundamental for data transmission between *TestSteps* involved in the test are defined within *TestScenarios*.

The values of *DataEntities* and *Variables* are defined in a tabular format with several rows. In this way, when an attribute is associated with *N* values, it means that this scenario may be executed *N* times, once for each value in the table.

Each scenario is also formed by *TestSteps*. A *TestStep* can have four types (*Actor_PrepareData, Actor_CallSystem, System_Execute, System_ReturnResult*) and several *OperationExtensions* (see Table 1).

It is in the test cases that are introduced the fundamental concepts for the test scripts generation, which include test scenarios and test steps. Listing 3 shows the test cases for the ”search of products in the store”.

In the *UseCaseTest* specification the respective *UseCase* and *DataEntities* specifications are associated and temporary variables are initialized, such as the name of the product that will be searched and the number of expected results. Also in the *UseCaseTest*, the *TestScenarios* are specified where the values are assigned to the variables and inserted the *TestSteps*, which contain the necessary information for the test scripts.

Listing 3: Example of ’Search Products’ test case TSL specification.

```
UseCaseTest t_uc_Search "Search Products" :
     Valid [
 useCase uc_Search actorInitiates aU_User
 description "As a User I want to search
     for a product by name or descripton"
 variable v1 [
  attribute search: String
  attribute expectedResults: String ]
 testScenario Search_Products :Main [
  isConcrete
  variable v1 withValues (
  | v1.search | v1.expectedResults +|
  | "Blouse"  | '1'   +|
  | "Summer"  | '3'    +|)
  step s1:Actor_CallSystem:Click element('
      Home')[\"The User clicks on the 'Home
      ' element\"]
  step s2:Actor_PrepareData:PostData
      readFrom v1.search ["The User writes
      'blouse' in the search text field"]
  step s3:Actor_CallSystem:Click button('
      Search_Product')["The User clicks on
      the 'Search' button"]
   step s4:System_Execute:Check
       elementOnScreen(limit v1.
       expectedResults)["The System checks
       if the number of results is the
       expected one"] ]
```

4.4 Generate Test Scripts

Once the specification is complete, it follows the generation of the test scripts for the Robot tool. This generation process is based on relations established between the RSL specification and the syntax of the Robot framework. It is possible to make an association of the the RSL concepts with the Robot framework syntax and some of the keywords made available by the Selenium library (Table 1).

First, in *Actor_PrepareData* type, it is expected that any type of data will be entered by the actor, such as text, passwords or even choose a file to upload. The value of the data to be entered is acquired through the *DataEntities* defined previously in the *TestScenario* when the *OperationExtension* of the *TestStep* is ‘readFrom’ followed by the identifiers of the respective *DataEntity/Variable* attributes.

Second, the *Actor_CallSystem* type associates the actions performed by the actor in the application, e.g., click a button, select checkbox. In *OperationExtension* the type of action (e.g., Click, Select, Mouse Over) and the element on which such action takes place (e.g., button, checkbox, image) are identified.

Third, there is the *System_ReturnResult* that is used when it is necessary to collect application data to

Table 1: Mapping between test case (RSL) and test scripts (Robot).

Step Type	Operation Extension Type	Operation Extension	Keyword generated
Actor_PrepareData	Input	readFrom	INPUT TEXT $locator $variable
Actor_CallSystem	Select	checkbox	SELECT CHECKBOX $locator
		list by value	SELECT FROM LIST BY VALUE $locator $value
	Click	button	CLICK BUTTON $locator
		element	CLICK ELEMENT $locator
	Over	-	MOUSE OVER $locator
System_ReturnResult	GetData	writeTo	$variable GET TEXT $locator
System_Execute	OpenBrowser	-	OPEN BROWSER $url
	CloseBrowser	-	CLOSE BROSER
	PostData	readFrom	INPUT TEXT $locator $variable
	Check	textOnPage	PAGE SHOULD CONTAIN $text
		elementOnPage	PAGE SHOULD CONTAIN ELEMENT $locator $msg? $limit?
		textOnElement	ELEMENT SHOULD CONTAIN $locator $text
		responseTime	WAIT UNTIL PAGE CONTAIN ELEMENT $locator $timeout?
		variableValue	$variable = $expected
		jScript	EXECUTE JAVASCRIPT $code

be stored in temporary variables that will normally be used for some type of verification. In this type of operation, the *OperationExtension* is 'writeTo' followed by the attribute of the variable where the value will be stored.

Finally, there is the *System_Execute* where the actions that are executed by the system, e.g., 'OpenBrowser' and 'Check', are associated. Each *TestScenario* must end with a Check in order to evaluate the success/insuccess of the test. The types of checks introduced are: text on element, element on page, text on element, response time, variable value or custom.

For instance, the code of the Listing 4 is generated from the test case of the Listing 3 through the mapping discussed.

At the end of this phase, the test scripts are created with the base structure of the Robot framework syntax and the keywords of the Selenium library.

Listing 4: Generated Test Script example (in Robot).

```
Search_Products-Test_1
  [Documentation] As a User I want to search
      for a product by name or descripton
  Click element [Home]
  Input text [Point&Click] ${search1}
  Click button [Search_Product]
  Page Should Contain Element [Point&Click]
      limit=${expectedResults1}

Search_Products-Test_2
  [Documentation] As a User I want to search
      for a product by name or descripton
  Click element [Home]
  Input text [Point&Click] ${search2}
  Click button [Search_Product]
  Page Should Contain Element [Point&Click]
      limit=${expectedResults2}
```

4.5 Map GUI Elements to Keywords

At this stage, there is the need to complete the test scripts generated in the previous phase with the locators (e.g. GUI element identifier, xpath) used for selecting the target GUI elements (Leotta et al., 2016). Web applications interfaces are formed by sets of elements, namely, buttons, message boxes, forms, among other elements that allow to increase the User Interface (UI) interactivity. Each of these elements has a specific locator, which allows it to be recognized among all elements of the UI. During the acceptance testing activity, these elements are used to achieve a certain position defined by the test case. In order to automate the acceptance test cases generation and execution, it is necessary to identify these locators to be able to use the respective GUI elements during the execution of the test.

Listing 5: Test Script with GUI elements xpath (in Robot).

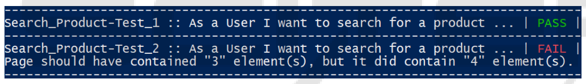
```
Search_Products-Test_1
 [Documentation]   As a User I want to
     search for a product by name or
     descripton
 Click Element //*[@id="header_logo"]/a/img
 Input text //*[@id="search_query_top"] ${
     search1}
 Click button //*[@id="searchbox"]/button
 Page should contain element //*[@id="
     center_column"]/ul/li limit=${
     expectedResults1}
Search_Products-Test_2
 [Documentation]   As a User I want to
     search for a product by name or
     descripton
 Click Element //*[@id="header_logo"]/a/img
 Input text //*[@id="search_query_top"] ${
     search2}
 Click button //*[@id="searchbox"]/button
 Page should contain element //*[@id="
     center_column"]/ul/li limit=${
     expectedResults2}
```

Since element identifiers usually do not follow any specific pattern, it becomes complex to do an automatic mapping. So, it is necessary the manual intervention of a technician to make the mapping between the GUI elements and the test scripts keywords. To establish this mapping the user can insert the corresponding identifiers of the UI elements in the test script or use a 'point and click' process (similar to the one in (Paiva et al., 2005)) where he points the UI element on screen to capture the identifier of the clicked element.

The elements of Listing 4 within the straight parentheses ([]) are semi-automatically replaced by the xpath of the web elements of the page to be tested resulting the test script in Listing 5.

4.6 Execute Tests

Once the script is completely filled in, the tests are run and the test results are displayed, as shown in Figure 5. In this test, when searching for 'Blouse', the test returned one result as expected and so, the test succeeded. On the other hand, when searching for 'Summer', the test returned 4 which is different from the expected result and so, the test failed.



```
------------------------------------------------------------------------------
Search_Product-Test_1 :: As a User I want to search for a product ... | PASS |
------------------------------------------------------------------------------
Search_Product-Test_2 :: As a User I want to search for a product ... | FAIL |
Page should have contained "3" element(s), but it did contain "4" element(s).
------------------------------------------------------------------------------
```

Figure 5: Result of the test case execution.

5 RELATED WORK

Acceptance test cases for complex IT systems are usually done manually and derived from functional requirements written in natural language. This manual process is challenging and time consuming.

Generating automatically test cases from textual or graphical models is not a new idea. Some approaches require that the system is captured by graphical models (e.g. worfkflow models (Boucher and Mussbacher, 2017) and domain models (Wang et al., 2015)) and others require textual models (e.g., use cases (Hsieh et al., 2013; Moketar et al., 2016)).

There is an approach that uses workflow notation focused on the casual relationship of the steps in a workflow without requiring the specification of detailed message exchange and data (Boucher and Mussbacher, 2017). These models are transformed into end-to-end acceptance test cases that can be automated with the Junit[4] testing framework. Comparing with the approach proposed in this paper, this technique does not allow the alignment between requirements and tests.

[4]https://junit.org/junit5/

Use Case Modeling for System Tests Generation (UMTG) is an approach that automatically generates system test cases from use case specifications and a domain model, the latter including a class diagram and constraints (Wang et al., 2015). Even though support test cases generation, this approach does not integrate any test execution automation tool associated to run the generated tests.

Hsieh et al. proposed the TestDuo framework for generating and executing acceptance tests from use cases (Hsieh et al., 2013). In this approach specific use case annotations are added by testers to explicate system behaviours. Similar to the approach presented, Robot compatible test cases are then generated. However, TestDuo does not cover the alignment between requirements and test cases.

TestMEReq is an automated tool for early validation of requirements (Moketar et al., 2016). This tool integrates semi-formalized abstract models called Essential Use Cases. Abstract test cases that describe the tested functionality of the requirements are generated from the abstract models, which helps to validate requirements. However, this approach does not contemplate the generated tests execution.

In contrast to the tools and approaches refer above, our proposed approach particularly promotes (i) the alignment between high-level requirements and tests specifications, with low-level test scripts, that is ensured by the adoption of languages like RSL and Robol, as well as (ii) semi-automatic generation and execution of test scripts, by the integration with tools like Robot framework.

6 CONCLUSION

This paper describes a model-based testing approach where acceptance test cases are derived from RSL requirements specifications and automatically adapted to the test automation Robot framework tool to be executed over a web application under test.

The process begins with the requirements elicitation and specification in the RSL. From these requirement specifications are generated test case specifications. When the test cases are complete, it is made the automatic generation of test scripts executable by the Robot framework. This generation is based on mappings between the characteristic constructs of RSL and the GUI elements identifiers of the SUT with the syntax of the Robot automation tool. Once test scripts are completed, they are executed and the results presented in a test report.

This approach allows to encourage the practice of testing when specifying requirements. In addition to reducing manual effort, time and resources dedicated to the development of tests, also ensures higher quality of requirements. The RSL requirements specification for the described generation will make the requirements specified more consistently and systematically and therefore less prone to errors and ambiguities.

As a future work, we intend to apply this approach in real context scenarios and automate further the overall process by automatically generating the test specification from RSL specification and converting these test specifications into executable test scripts that may be executed by other test automation tools, such as Gherkin/Cucumber[5].

ACKNOWLEDGEMENTS

This work was partially supported by national funds under FCT projects UID/CEC/50021/2019 and 02/SAICT/2017/29360.

REFERENCES

Bhat, A. and Quadri, S. (2015). Equivalence class partitioning and boundary value analysis - A review. *2nd International Conference on Computing for Sustainable Global Development (INDIACom)*.

Boucher, M. and Mussbacher, G. (2017). Transforming Workflow Models into Automated End-to-End Acceptance Test Cases. *Proceedings - 2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering, MiSE 2017*, pages 68–74.

Christie, J. (2008). Test automation - does it make sense? *TE - Testing Experience, The Magazine for Professional*, pages 73—77.

Cockburn, A. (2001). *Writing Effecive Use Cases*. Addison-Wesley.

da Silva, A. R. (2017). Linguistic patterns and linguistic styles for requirements specification (i): An application case with the rigorous rsl/business-level language. In *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, pages 22:1–22:27, New York, NY, USA. ACM.

Hsieh, C. Y., Tsai, C. H., and Cheng, Y. C. (2013). Test-Duo: A framework for generating and executing automated acceptance tests from use cases. *2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings*, pages 89–92.

ISTQB (2014). ISTQB ® Foundation Level Extension Syllabus Agile Tester.

ISTQB (2015). ISTQB ® Foundation Level Certified Model-Based Tester Syllabus.

Jacobson, I. and et al (2015). Object oreinted software engineering: A use case driven approach. *Addison-Wesley*.

Kovitz, B. (1998). Pratical software requirements: Manual of content and style. manning.

Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2016). *Approaches and Tools for Automated End-to-End Web Testing*, volume 101. Elsevier Inc., 1 edition.

Moketar, N., Kamalrudin, M., Sidek, S., Robinson, M., and Grundy, J. (2016). TestMEReq: Generating abstract tests for requirements validation. *Proceedings - 3rd International Workshop on Software Engineering Research and Industrial Practice, SER and IP 2016*, pages 39–45.

Moreira, R. M. and Paiva, A. C. (2014). 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*, ASE '14, pages 863–866, New York, NY, USA. ACM.

Moreira, R. M. L. M., Paiva, A. C. R., Nabuco, M., and Memon, A. (2017). Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Softw. Test., Verif. Reliab.*, 27(3).

Paiva, A. C. R. (2007). *Automated Specification-based Testing of Graphical User Interfaces*. PhD thesis, Faculty of Engineering of the University of Porto, Porto, Portugal.

Paiva, A. C. R., Faria, J. C. P., Tillmann, N., and Vidal, R. F. A. M. (2005). A model-to-implementation mapping tool for automated model-based GUI testing. In *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, pages 450–464.

Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, First edition.

Robertson, S. and Robertson, J. (2006). *Mastering the Requirements Process*. Addison-Wesley, 2nd edition edition.

Silva, A. R. (2018). Rigorous Requirements Specification with the RSL Language: Focus on Uses Cases. *INESC-ID Technical Report*.

Silva, A. R., Paiva, A. C., and Silva, V. (2018). Towards a Test Speccification Language for Information Systems: Focus on Data Entity and State Machine Tests. *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*.

Wang, C., Pastore, F., Goknil, A., Briand, L., and Iqbal, Z. (2015). Automatic Generation of System Test Cases from Use Case Specifications. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 385–396.

Withall, S. (2007). *Software Requirements Patterns*. Microsoft Press.

[5]https://cucumber.io/