From Requirements to Automated Acceptance Tests with the RSL Language

Ana C. R. Paiva[1,2](✉), Daniel Maciel[1], and Alberto Rodrigues da Silva[3]

[1] Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
apaiva@fe.up.pt
[2] INESC TEC, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
daniel.ademar.maciel@gmail.com
[3] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
alberto.silva@tecnico.ulisboa.pt

Abstract. Software testing can promote software quality. However, this activity is often performed at the end of projects where failures are most difficult to correct. Combining requirements specification activities with test design at an early stage of the software development process can be beneficial. One way to do this is to use a more structured requirements specification language. This allow to reduce typical problems such as ambiguity, inconsistency, and incorrectness in requirements and may allow the automatic generation of (parts of) acceptance test cases reducing the test design effort. In this paper we discuss an approach that promotes the practice of requirements specification combined with testing specification. This is a model-based approach that promotes the alignment between requirements and tests, namely, test cases and also low-level automated test scripts. To show the applicability of this approach, we integrate two complementary languages: (i) the ITLingo RSL (Requirements Specification Language) that is specially designed to support both requirements and tests rigorously and consistently specified; and (ii) the Robot language, which is a low-level keyword-based language for specifying test scripts. This approach includes model-to-model transformation processes, namely a transformation process from requirements (defined in RSL) into test cases (defined in RSL), and a second transformation process from test cases (in RSL) into test scripts (defined according the Robot framework). This approach was applied in a fictitious online store that illustrates the various phases of the proposal.

Keywords: Requirements Specification Language (RSL) · Test case specification · Model-based Testing (MBT) · Test case generation · Test case execution

1 Introduction

Software systems are becoming increasingly complex and operating on more critical systems. This reality makes it more urgent to run software tests that promote the quality of these systems. One aspect of software quality is its ability to meet the implicit and explicit needs of customers. For this, it is important to reach a common understanding between clients and developers about what should be developed.

Requirements Engineering (RE) helps to create the basis of understanding between stakeholders and programmers about the software system to develop. The resulting system requirements specification (SRS) document helps to structure the view on the software system and allows [1–4] to agree between users and developers on the validation and verification support of the scope of the project and support future system maintenance activities. The problem is that the manual effort required to produce requirements specifications is high and suffers from problems such as incorrectness, inconsistency, incompleteness and ambiguity [2,3,6].

ITLingo is a long-term initiative aimed at researching, developing and applying rigorous IT specification languages, i.e., Requirements Engineering, Test Engineering and Project Management [22]. ITLingo takes a linguistic approach to improve the accuracy of technical documentation (e.g., SRS, test case specification, project plans) and, as a consequence, promote productivity through reuse and model transformations as well as promote systems quality through semi-automatic validation techniques.

Requirements Specification Language (RSL) is a controlled and integrated natural language with ITLingo that assists in the production of requirements specifications in a systematic, rigorous and consistent manner [5]. RSL includes an advanced set of constructs that are logically organized into views according to specific RE concerns at different levels of abstraction, such as business, applications, software, or even hardware levels.

Software testing can also be useful as a measure for assessing the software development process by measuring the number of tests that pass or fail and conducting regression tests to foster product quality by alerting developers to potential defects as soon as code is changed.

Acceptance tests are those that are most closely related to requirements as they reflect what the end user considers important to test (needs, requirements and business processes) to accept or not the software that is being developed [25].

To reduce the time and resources required, it may be helpful to perform acceptance test design and specification requirements in parallel [11]. Although it is considered a good practice to start testing activities at the beginning of the project when requirements are elicited, this does not always happen because elicitation and requirements testing are separate in traditional development processes. This research paper presents an approach based on the Model-Based Testing (MBT) technique [25] that aims to foster the initiation of testing activities early in line with the requirements specification. MBT is a software testing approach that generates test cases from abstract representations of the system, named models, either graphical (e.g., Workflow models [16], PBGT [19,20]) or textual (e.g., requirements documents in an intermediate format) [24].

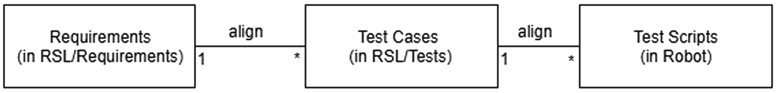


Fig. 1. Key structural concepts [28].

The process (Fig. 1) starts by producing RSL *Requirements* specifications based on the set of constructs provided by the language and according to different perspectives and concerns. From those *Requirements*, it is possible to generate RSL *Test Cases* specifications, and from these, to generate *Test Scripts*. Finally, those test scripts can be automatically executed by the Robot Test Automation Tool[1] in the application under test.

This paper extends [28] in the following aspects:

– It extends Sect. "2.2 – Tests Specification" by detailing the grammar of RSL in what concerns the following constructs: $UseCaseTest$, $TestScenario$, $TestStep$, $TestOperation$ and $TestCheck$.
– It adds new figures: one to illustrate the RSL/Tests Extension and another to illustrate the Mapping process between GUI elements and keywords.
– It restructures Sect. 4 by splitting it into two sections: Sect. 4 to describe the overall approach; Sect. 5 called "5 - Illustrative Example" where it illustrates, in more detail, the applicability of the overall approach over a fictitious online store developed to practice and validate the test automation.
– It extends the state of the art.

This paper is organized in 7 sections. Section 2 overviews the RSL language, showing its architecture, levels of abstraction, concerns and grammar. 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 presents a case study illustrating the overall approach. Section 6 identifies and analyzes related work. Finally, Sect. 7 presents the conclusion and 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 (Requirements Specification Language) [7–10,17,18]. RSL provides a vast set of logically organized constructs in views that describe different concerns. These constructs are defined by *linguistic patterns* which are represented textually according to concrete *linguistic styles* [5]. RSL can be used and adapted by different organizations because it is a process and tool independent language [5,22]. The constructs used by RSL can be classified according to two perspectives [22]: concerns and abstraction levels. The concerns are: active structure (subjects), behaviour (actions), passive structure (objects), requirements, tests, other concerns, relations and sets. The abstraction levels are: business, application, software and hardware levels. This paper focuses the discussion on the requirements and tests concerns and, in particular, focuses on the RSL constructs particularly supportive of use case approaches (e.g. actors, use cases, data entities and involved relationships) as it is further discussed in [22].

[1] http://robotframework.org/.

2.1 Requirements Specification

Figure 2 shows a part of the RSL metamodel. It defines the hierarchy established among requirement types, namely: goal, functional requirement, constraint, use case, user story and quality requirement. This paper focuses only on the discussion of *UseCase* requirement and test types.

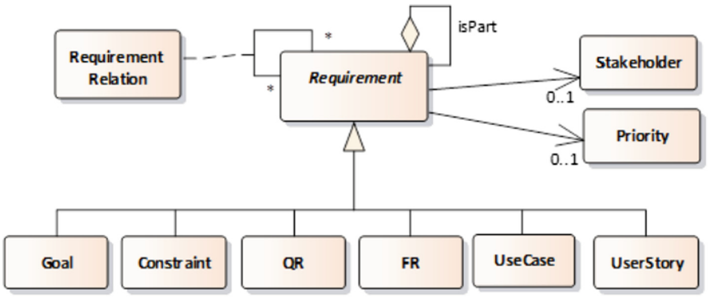


Fig. 2. RSL partial metamodel: the hierarchy of requirements [28].

RSL specifications based on *Use Cases* may involve defining some views with their inherent constructs and relationships, namely:

- *DataEntity view*: defines the structural entities within an information system, commonly associated with data concepts captured and identified from domain analysis. A *Data Entity* denotes an individual structural entity that may include specifying attributes, foreign keys, and other verification data constraints;
- *DataEntityCluster view*: denotes a cluster of various structural entities that have a logical arrangement with each other;
- *Actor view*: defines the participants of *Use Cases* or *user stories*. They represent end users and external systems that interact directly with the system under study and, in some particular situations, may represent timers or events 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 on a system to achieve a specific outcome [12].

2.2 Tests Specification

RSL supports Test Cases specification and generation directly from the requirements specifications. As shown in Fig. 3, RSL provides an hierarchy of Test constructs that supports specifying the following test case specializations [22]:

- *DataEntityTest* are obtained from equivalence class partitioning and boundary value analysis techniques applied over the domains defined for the *DataEntities* [23] in RSL *DataEntities*;

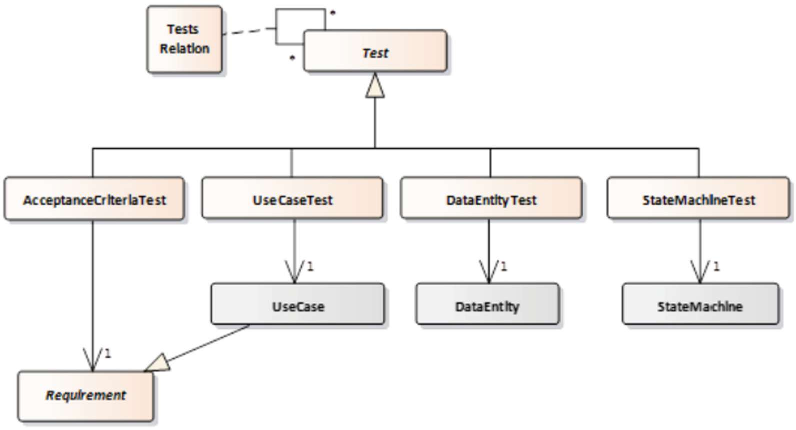


Fig. 3. RSL partial metamodel: the hierarchy of Tests [28].

- *UseCaseTest* explores different sequences of steps defined in RSL *use cases*' scenarios, and associates data values to the involved *DataEntities*;
- *StateMachineTest* applies different algorithms to traverse RSL state machines so that different test cases can be defined that correspond to valid or invalid paths through their state machine;
- *AcceptanceCriteriaTest* defines acceptance criteria based on two distinct approaches: scenario based (i.e., based on the 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 set to Valid or Invalid depending on the intended situation. In addition, it is possible to establish relationships with other test cases through *TestsRelation*; these relationships can be classified as *Requires*, *Supports*, *Obstructs*, *Conflicts*, *Identical*, and *Relates*.

With respect to the different RSL Test constructs described, *UseCaseTests* best fit the acceptance test. Figure 4 shows the structure and relationships of *UseCaseTests*. A $UseCaseTest$ (Listing 1.1) inherits $UseCase$ data associated with it, including $Actors$. Optionally, it is possible to add variables for testing purposes as well.

An $UseCaseTest$ may have different $TestScenarios$ (Listing 1.2). Each scenario must have, at least, one $TestStep$ and, if necessary, values assigned to $DataEntities$ and $Variables$. Since $DataEntities$ are entities of the Application Under Test, it may be useful to create instances of these entities and assign them values that may be used later in test cases.

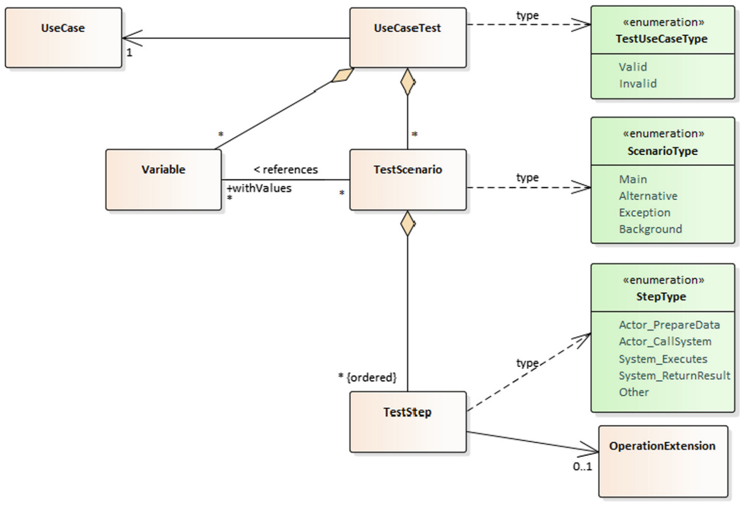


Fig. 4. RSL/tests extension.

Listing 1.1. UseCaseTest RSL grammar [31].

```
UseCaseTest:
'UseCaseTest' name=ID (nameAlias=STRING)? ':' type=TestType ('['
'useCase' useCase=[UseCase | QualifiedName]
('actorInitiates' actorInitiates=[Actor | QualifiedName] )
('actorParticipates' actorParticipates+=RefActor)?
('background' background=[UseCaseTest | QualifiedName] )?
(variables+=TestVariable)∗
(scenarios+=TestScenario)∗
(tags+=Tag)∗
('description' description=STRING)?
']')?;
```

Variables are temporary data that may be exchanged among $TestSteps$, e.g., a variable may be used to save text needed to validate the dynamic content on the GUI.

Listing 1.2. TestScenario RSL grammar [31].

```
TestScenario:
'testScenario' name=ID (nameAlias=STRING)? ':' type=ScenarioType ('['
((isConcrete ?= 'isConcrete') | (isAbstract ?= 'isAbstract'))?
('variable'variable= [TestVariable | QualifiedName] ('withValues' '('
variableTable= DataVariableValues ')'))?
('dataEntity' entity= [DataEntity | QualifiedName] ('withValues' '(' entityTable=
DataAttributeValues ')'))?
('executionMode' mode=('Sequential'|'Parallel'))?
('description' description=STRING)?
testSteps+= TestStep+
']')?;
```

The $TestStep$ (Listing 1.3) is classified by a $StepOperationType$ and, optionally, by a $StepOperationSubType$. The operation types are used to define the action that are performed in each step.

Listing 1.3. TestStep RSL grammar [31].

```
TestStep:
'step' name=ID ':' type=StepOperationType (':' extension=OperationExtension)? ('['
(simpleTestStep= SimpleTestStep );

OperationExtension:
(subType=StepOperationSubType)
((target=TestOperationTarget)|(check=TestCheck))?;

enum StepOperationType: Actor_PrepareData | Actor_CallSystem | System_Execute | System_ReturnResult | Other |
        None;
enum StepOperationSubType: OpenBrowser | CloseBrowser | Reload | GetData | PostData | Select | Click | Over |
        Check | Other;
```

There are four general types of operations performed in $TestSteps$ [30]:

- $Actor\_PrepareData$: input data will be entered by the actor, such as text, passwords or even choose a file to upload;
- $Actor\_CallSystem$: actions performed by the actor in the application, e.g., click a button, select checkbox;
- $System\_ReturnResult$: collect application data to be stored in temporary variables. This is usually helpful to perform some type of verification;
- $System\_Execute$: actions executed by the system, e.g., open the browser and validations.

The $StepOperationSubTypes$ are an extension of the previous types specifying the operations performed. These sub types are [31]:

- $Open/CloseBrowser$: action to open/close the browser;
- $Reload$: action to reload the browser page;
- $GetData$: action to collect specific data from the AUT;
- $PostData$: action to post specific data to the AUT;
- $Select/Click/Over$: to specify the action to be performed in an AUT element;
- $Check$: action to verify some AUT content or response; Each step operation must have a target (*TestOperationTarget*) or a verification ($TestCheck$) depending on the action associated (Listing 1.4).

If the action is an interaction with a GUI element, the $TestOperationTarget$ (Listing 1.4) will specify that element through the $OperationTargetType$. It can be a button, a generic element, a checkbox or a list. Additionally, the $OperationTargetType$ may also be used to clarify if such element is used to “write to” or to “read from”. Finally, the $TestOperationTarget$ can also have a description that is sent as a parameter through a variable value or a string.

Listing 1.4. TestOperation and TestCheck RSL grammar [31].

```
TestOperationTarget:
(type=OperationTargetType)
((variable+=[DataAttribute | QualifiedName] (‘,’variable+=[DataAttribute |
QualifiedName] )∗)|
(‘(’ content+=(STRING) (‘,’content+=STRING)∗ ‘)’))?;
enum OperationTargetType : button | element | checkbox | listByValue | readFrom |
writeTo;

TestCheck:
(type=CheckType) (‘(’
(variable=[DataAttribute | QualifiedName] ‘=’ expected=[DataAttribute |
QualifiedName])?
(‘text’ (textVariable=[DataAttribute | QualifiedName]| textString=STRING))?
(‘timeout’ (timeoutVariable=[DataAttribute | QualifiedName]| timeoutINT=
DoubleOrInt) metric=Metric?)?
(‘limit’ (limitVariable=[DataAttribute | QualifiedName]| limitINT=INT))?
(‘url’ (urlVariable=[DataAttribute | QualifiedName]| urlString=STRING))?
(‘code’ (codeVariable=[DataAttribute | QualifiedName]| codeString=STRING))?
‘)’);
enum CheckType: textOnScreen | textOnElement | elementOnScreen | responseTime | variableValue | script | screen |
      Other | None;
```

The $TestCheck$ defines the validation to perform in the step where it was specified. There are seven types of validations ($CheckTypes$) and each of them has different parameters. Table 1 shows the set of validations available. Each $TestScenario$ must end with a $TestStep$ that has a $TestCheck$. If the check succeeds the test passes. If the check does not succeeds, the test fails.

Table 1. Test step validations [31].

CheckType	Parameter	Validation
textOnScreen	text	checks if a specific text is presented in the GUI
textOnElement	text	checks if a specific text is presented in a specific element of the GUI
elementOnScreen	limit?	checks if a specific element is presented in the GUI. If a limit is sent as parameter checks if a specific element appears less than the limit established
responseTime	timeout	checks if the response time is less or equal than the given timeout
variableValue	variable	checks if a variable value is equal to the expected value
	expected	
screen	URL	checks if the page represents the given URL
script	Code	uses a custom script to validate an unusual case

3 Robot Framework

Test cases can be run manually by the tester or automatically by a test automation tool. When a test case is run manually, the tester must execute all test cases and have 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. Therefore, if a test case is to be run multiple 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, which includes out-of-the-box libraries. The robot does not require any implementation as it is possible to use keywords with implicit implementations (using specific libraries such as Selenium[2]). Robot is open source and related to acceptance test-driven development (ATDD) [27]. It is operating system independent and is implemented natively in Python and Java, and can be run on Jython (JVM) or IronPython (.NET).

The structure of the script is simple and can be divided into four sections. The first section, *Settings*, where the paths to helper files and libraries used are set. The second section, *Variables*, specifies the list of variables used as well as the associated values. The third and most important section is the *Test Cases*, where test cases are defined. Finally, the *Keywords* section defines custom keywords to implement the test cases described in the Test Cases section. Among all four sections, only *Test Cases* is mandatory.

As seen in the example shown in Listing 1.5, the libraries used are initially defined. One of the most widely used is the Selenium library, which introduces keywords related to interactive application testing, such as ‘Open Browser’ and ‘Input text’. The variables section assigns ‘Blouse’ to the variable ‘product’ so whenever ‘product’ is used it has the value ‘Blouse’. The Keywords section defines keywords and their parameters. In test cases that use keywords, the values are assigned to the corresponding parameters, placing the values in the same place where the parameters are set.

Listing 1.5. Robot Framework specification example [28].

```
*** 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

Although it is considered a good practice to start testing activities early in the project, this is not frequently the common situation due to the traditional separation between the requirements and testing phases. This research intends to reduce this problem through a

[2] www.seleniumhq.org/.

framework that encourages and supports 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 (defined in Fig. 5) 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.

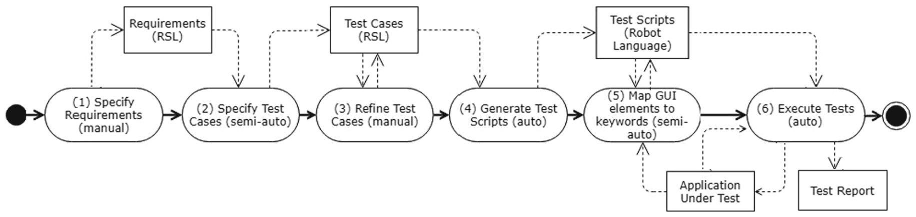

Requirements (RSL)
Test Cases (RSL)
Test Scripts (Robot Language)
(1) Specify Requirements (manual)
(2) Specify Test Cases (semi-auto)
(3) Refine Test Cases (manual)
(4) Generate Test Scripts (auto)
(5) Map GUI elements to keywords (semi-auto)
(6) Execute Tests (auto)
Application Under Test
Test Report

Fig. 5. Proposed approach (UML activity diagram) [31].

This set of tasks covers the process of acceptance tests in interactive applications from the specification of requirements to the execution of tests. Applying the approach will establish an alignment between the specification of requirements and the specification of tests, in addition to increasing the processes automation. Besides the use and extension of the RSL grammar, the approach also uses support tools such as the Robot framework and Web Scrapper.

4.1 Specify Requirements

The first task of this approach is the requirements definition, an activity that usually involves the intervention of requirements engineers, stakeholders and eventually testers. After reaching a consensus, the specification of the requirements in RSL follows, through the constructs provided by the language that most fit the requirements domain. In this approach, the specification focuses on the most relevant RSL constructs at the application and software level, namely: $Actor$, $UseCase$, $DataEntity$ and involved relationships. This task is usually performed by business analysts or by requirement engineers.

4.2 Specify Test Cases

$UseCaseTests$ are derived from the various process flows expressed by a RSL $UseCase$. Each test contains multiple test scenarios which encompasses of a group of test steps. From the requirement specifications, it is possible to specify test cases. $UseCaseTest$ construct begins by defining the test set, including ID, $name$ and the $usecasetype$. Then it encompasses the references keys $[UseCase]$ indicating the Use Case in which the test is proceeding and $[DataEntity]$ referring to a possible data entity that is managed.

In the $UseCaseTest$ specification, the respective $UseCase$ and $DataEntities$ specifications are associated, temporary variables are initialized, the $TestScenarios$

are specified where values are assigned to the variables and $TestSteps$ are inserted which contain the necessary information for the test scripts.

4.3 Refine Test Cases

Generated test cases may be refined manually (e.g., assign values to entities and create temporary variables), which results in other test cases.

The information introduced in the requirements specification phase and the RSL constructs allow to simplify the test cases construction.

The $DataEntities$ and the temporary $Variables$ are fundamental for transmitting data between the $TestSteps$ involved in the test and are defined within $TestScenarios$.

The values of $DataEntities$ and $Variables$ may be defined in table. By using this table structure, when an attribute is associated with N values, the test scenario may be executed N times (one time for each value in the table).

4.4 Generate Test Scripts

Once the specification is complete, the generation of the test scripts for the Robot tool follows. This generation process is based on relations established between the RSL specification and the syntax of the Robot framework. An association of the RSL concepts with the Robot framework syntax and some of the keywords made available by the Selenium library are shown in Table 2.

Table 2. Mapping between test case (RSL) and test scripts (Robot) [28].

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

4.5 Map GUI Elements to Keywords

At this phase of the process, there is the need to complete the test scripts generated previously with the locators [26], i.e. queries that return a single GUI element which are used to locate the target GUI elements (e.g., GUI element identifier, xpath, CSS selector).

This mapping can be established by the user inserting directly the identifiers of the UI elements in the test script, or by using a 'point and click' process (similar to the one presented in [21]). In this last option Table 2, the user accesses the AUT and, with the help of the Web Scrapper, points to the desired elements.

The Web Scrapper saves the locators (unique CSS selector) and exports this information to JSON code. Then, the tester can execute the Mapping script that creates a XML file with the relation between locators found and the ones missing from the Robot test script generated previously. For this process to succeed, it is important to maintain consistency between the descriptions of the elements in both Web Scrapper and test case specification. Finally, the tester may execute the Replacement Script that completes the Robot Script with the locators based on the data provided in the XML file.

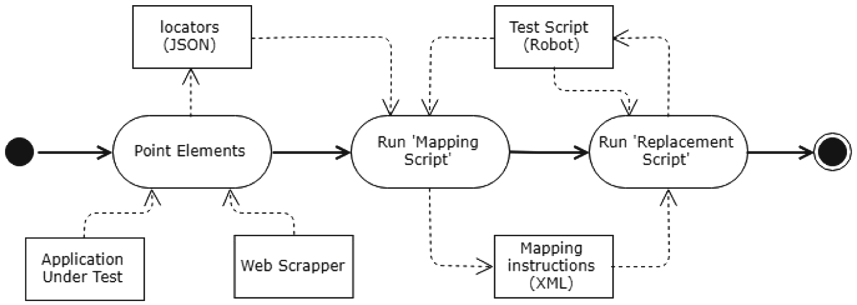


Fig. 6. Map GUI Elements to keywords (UML activity diagram) [31].

4.6 Execute Tests

The generated test script is executed by using the Robot framework. For that, the user should use, at the command prompt, following command:

$robot[script\_name].robot$

During execution, a browser instance will open performing automatically every steps specified in the test script while showing the results of each test case at the command prompt.

5 Illustrative Example

In order to illustrate and discuss the suitability of the proposed approach, we show its application with an interactive web application as the application under test (AUT). We selected the “Web Store”[3] app: This is a popular e-commerce web site developed on purpose to practice test automation. It simulates common online shopping workflows. Figure 7 shows the home page of this online store.

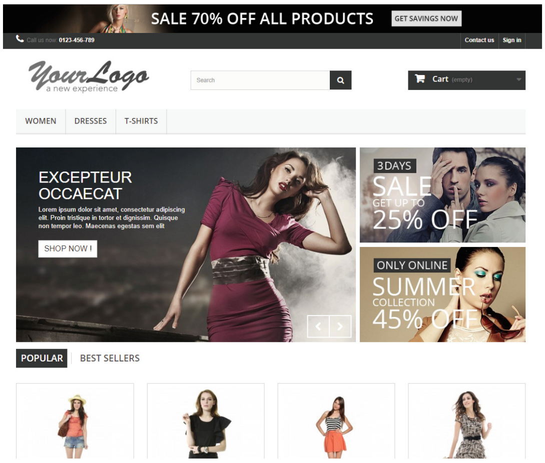


Fig. 7. Web Store application - Search Product [31].

We consider the use case “Search Product” because it is a simple and illustrative example. In this use case, the user searches for a product by its name and the number of items matched must be equal to the expected one defined in the respective test case.

After the requirements specification, that is partially shown in Listing 1.6, it follows the definition of test cases, where the relationships between the (use case) requirement and the (use case) tests are kept. A $UseCaseTest$ is generated or manually created based and aligned with the corresponding $UseCase$. After that, the test case can be refined with $TestScenarios$, $TestSteps$ and references to involved $DataEntities$ and $Variables$.

Listing 1.6. Example of a RSL specification of DataEntity Actor and UseCase [28].

```
DataEntity e_Product ‘‘Product’’ : Master [
    attribute ID ‘‘ID’’ : Integer [isNotNull 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 [
    description ”Customer uses the system”]

UseCase uc_Search ‘‘Search Products’’ : EntitiesSearch [
    actorInitiates aU_Customer
    dataEntity e_Product]
...
```

[3] http://automationpractice.com.

Listing 1.7 shows a test case specified and refined with the necessary information to define such tests. In this case, two variables were associated. The first one, $v1.search$, is the keyword used to input the name of the search products; the second variable, $v1.expected$, is used to define the number of results expected.

Listing 1.7. Example of 'Search Products' test case RSL specification [28].

```
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 a word or phrase 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'' ] ]
```

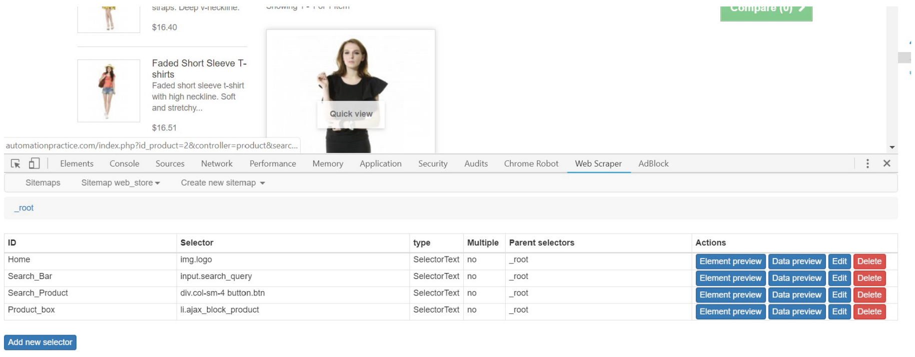
In the $TestScenario$ (Listing 1.7) it is defined the ordered steps that are necessary to perform the actions to get the number of search results and compare it with the expected number.

Once the test case specification is completed, it follows the generation of the equivalent test scripts for the Robot framework: that code generator (integrated in the ITLingo-Studio) generates a set of test scripts (in Robot language), resulting in a script similar to the one shown in Listing 1.8. However, there still miss the elements locators specified in the script, so that the Robot framework could know in which concrete elements of the AUT it shall perform the command specified by the test script.

Listing 1.8. Generated Test Script example (in Robot) [28].

```
*** Variables ***
${search1} Blouse
${search2} Summer
${expectedResults1} 1
${expectedResults2} 4

*** Test Cases ***
Search_Products-Test_1
   [Documentation] As a User I want to search for a product by name or descripton
   Click element [Home]
   Input text [Search_Bar] ${search1}
   Click button [Search_Product]
   Page Should Contain Element [Product_box] limit=${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 [Search_Bar] ${search2}
   Click button [Search_Product]
   Page Should Contain Element [Product_box] limit=${expectedResults2}
```

To provide this missing information, it is necessary to map GUI elements with appropriate keywords as suggested in Fig. 6.



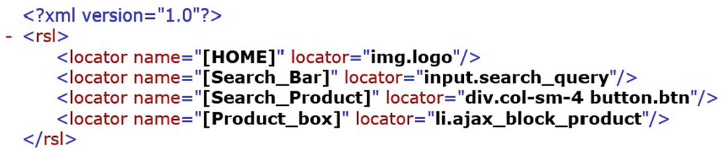
ID	Selector	type	Multiple	Parent selectors	Actions
Home	img.logo	SelectorText	no	_root	Element preview Data preview Edit Delete
Search_Bar	input.search_query	SelectorText	no	_root	Element preview Data preview Edit Delete
Search_Product	div.col-sm-4 button.btn	SelectorText	no	_root	Element preview Data preview Edit Delete
Product_box	li.ajax_block_product	SelectorText	no	_root	Element preview Data preview Edit Delete

Fig. 8. Web Scrapper [31].

For that purpose the tester shall access the AUT and shall point to the desired elements with the help of the Web Scrapper (Fig. 8). The information of every locators is exported to the JSON file. After executing the Mapping script, the XML file (Fig. 9) is generated with the information about the missing locators of the previous phase. Finally, the replacement scripts fills in the missing information resulting in a concrete script as illustrated in Listing 1.9. In this Listing it is possible to see that $css : img.logo$ is the CSS locator for the element that redirects the user to the "Home Page". Once complete, the test script will be able to be executed.

Listing 1.9. Test Script with GUI elements xpath (in Robot) [31].

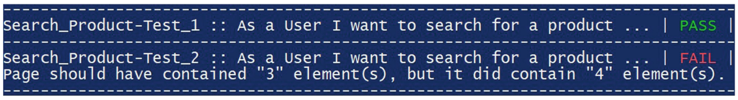
```
*** Variables ***
${search1} Blouse
${search2} Summer
${expectedResults1} 1
${expectedResults2} 4

*** Test Cases ***
Search_Product−Test_1
  [Documentation] As a User I want to search for a product by name or descripton
  Click Element css:img.logo
  Input text css:input.search_query ${search1}
  Click button css:div.col−sm−4 button.btn
  Page should contain element css:li.ajax_block_product limit=1

Search_Product−Test_2
  [Documentation] As a User I want to search for a product by name or descripton
  Click Element css:img.logo
  Input text css:input.search_query ${search2}
  Click button css:div.col−sm−4 button.btn
  Page should contain element css:li.ajax_block_product limit=3
```



```
<?xml version="1.0"?>
- <rsl>
    <locator name="[HOME]" locator="img.logo"/>
    <locator name="[Search_Bar]" locator="input.search_query"/>
    <locator name="[Search_Product]" locator="div.col-sm-4 button.btn"/>
    <locator name="[Product_box]" locator="li.ajax_block_product"/>
  </rsl>
```

Fig. 9. XML file – Map between locators and test script [31].

Once the script is completely filled in, these test scripts can be executed and such test results can be obtained, as shown in Fig. 10. Regarding the use case "product search", when searching for products associated to the word $Blouse$ (Test_1), the test returned "1" which is the expected result and so, the test succeeded. On the other hand, when searching for products related to the word $Summer$ (Test_2), the test returned "4" products which is different from the expected result ("3") 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).
------------------------------------------------------------------------------
```

Fig. 10. Result of the test case execution [28].

6 Related Work

It is common to derive acceptance test cases for complex IT systems manually from functional requirements described in natural language. This manual process is challenging and time consuming.

One way to diminish this effort is to generate test cases automatically from textual or graphical models. This is not a new idea. In fact, there are some approaches that require graphical models (e.g. workflow models [16], or domain models [13]) or, others, requiring textual models (e.g., use cases [14, 15]) of the system.

The approach followed by [16] uses a workflow notation in which the focus is the casual relationship of the steps without specification of detailed message exchange and data. From these models it is possible to generate end-to-end test cases that are automated using the Junit[4] testing framework. This approach does not align requirements and tests like the one described in this paper.

In [13], the UMTG (Use Case Modeling for System Tests Generation) approach generates automatically system test cases from use case specifications and domain models (class diagram and constraints). This research work does not include any test execution automation tool to run the generated tests.

Hsieh et al. [14] proposed the Test-Duo framework for generating (and executing) acceptance tests from use cases. The testers add specific use cases annotations to clarify the system behaviour. The final test scripts are compatible with Robot framework. However, this approach does not align requirements with tests.

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

TestMEReq is an automated tool for early validation of requirements [15] described by semi-formalized abstract models called Essential Use Cases. From these abstract models it generates abstract test cases to help validate the requirements. This approach does not include the execution of the generated abstract test cases.

In [29], the authors present the design of a test automation platform, ETAP-Pro, to test end-to-end business processes that aims to overcome some challenges in validating business processes. ETAP-Pro works over BPMN models and is based on a keyword-driven approach. It generates test cases specifications in Gherkin. This approach promotes alignment between test cases and requirements since it maintains traceability information among test cases, requirements and keywords. However, test scripts should be generated manually to be executed afterwords.

In contrast with some tools and approaches mentioned above, our proposal particularly promotes the alignment between high-level requirements and tests specifications, and with low-level test scripts, that is ensured by the adoption of languages like RSL and Robot. In addition, this proposal promotes the quality and productivity of both (Requirements an Testing) tasks by considering model-to-model transformation features (e.g., RSLfrom Requirements into RSL Test Cases, or from RSL Test Cases into Robot Test Scripts), and execution of Robot Test Scripts, with the integration of tools like ITlingo-Studio and Robot framework.

7 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 to be executed against a web application under test.

This process begins with the requirements elicitation and specification in the RSL. From these requirement specifications (defined in RSL) are created manually or generated test case specifications (also in RSL), which are strongly kept aligned. When these test cases are completely defined, a second model-to-model transformation process is performed, which produces quasi-executable test scripts (in Robot language), which needs to be mapped to concrete GUI elements before be executable by the Robot framework. This generation is based on mappings between the characteristic constructs of RSL and the GUI elements identifiers of the AUT with the syntax of the Robot automation tool. Once test scripts are completed, they are executed and the results presented in a test execution report.

This approach encourages the practice of specifying both requirements and tests during the early stages of the projects, and keeping these specifications aligned with each other. It also promotes the productivity by reducing manual effort, time and resources dedicated to the development of tests, also ensures higher quality of requirements. The adoption of a language like RSL, that supports both requirements and tests specification in a more consistent and systematic way, is therefore less prone to errors and ambiguities.

As future work, we intend to extensively apply this approach in both controlled and real-world scenarios. We also intend to further improve the productivity of the proposed approach by automatically generating RSL test specifications from RSL requirement specifications (e.g., considering other types of test cases and other situations like security or performance) and generating these test specifications into executable test scripts that may be executed by multiple test automation frameworks, 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

1. Cockburn, A.: Writing Effective Use Cases, 1st edn. Addison-Wesley, Boston (2000)
2. Kovitz, B.L.: Practical Software Requirements: Manual of Content and Style. Manning Publications, Greenwich (1998)
3. Robertson, S., Robertson, J.: Mastering the Requirements Process: Getting Requirements Right, 3rd edn. Addison-Wesley Professional, Boston (2012)
4. Withall, S.: Software Requirements Patterns, 1st edn. Microsoft Press (2007)
5. Silva, A.R.: 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. ACM (2017)
6. Pohl, K.: Requirements Engineering: Fundamentals, Principles, and Techniques, 1st edn. Springer, Heidelberg (2010)
7. Ferreira, D.A., Silva, A.R.: RSLingo: an information extraction approach toward formal requirements specifications. In: 2nd IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012 - Proceedings, pp. 39–48 (2012)
8. Videira, C., Ferreira, D., Silva, A.R.: A linguistic patterns approach for requirements specification. In: Proceeding 32nd Euromicro Conference on Software Engineering and Advanced Applications (Euromicro 2006). IEEE Computer Society (2006)
9. Ferreira, D.A., Silva, A.R.: RSL-PL: a linguistic pattern language for documenting software requirements. In: 3rd International Workshop on Requirements Patterns, RePa 2013 - Proceedings, pp. 17–24 (2013)
10. Ferreira, D.A., Silva, A.R.: RSL-IL: an interlingua for formally documenting requirements. In: 3rd International Workshop on Model-Driven Requirements Engineering, MoDRE 2013 - Proceedings, pp. 40–49 (2013)
11. Silva, A.R., Saraiva, J., Ferreira, D., Silva, A.R., Videira, C.: Integration of RE and MDE paradigms: the ProjectIT approach and tools. IET Softw. **1**, 294–314 (2007)
12. Jacobson, I., et al.: Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Boston (2015)
13. Wang, C., Pastore, F., Goknil, A., Briand, L., Iqbal, Z.: Automatic generation of system test cases from use case specifications. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 385–396 (2015)
14. Hsieh, C., Tsai, C., Cheng, Y.C.: Test-duo: a framework for generating and executing automated acceptance tests from use cases. In: 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings, pp. 89–92 (2013)
15. Moketar, N.A., Kamalrudin, M., Sidek, S., Robinson, M., Grundy, J.: TestMEReq: generating abstract tests for requirements validation. In: Proceedings - 3rd International Workshop on Software Engineering Research and Industrial Practice, SER and IP 2016, pp. 39–45 (2016)

[5] https://cucumber.io/.

16. Boucher, M., Mussbacher, G.: Transforming workflow models into automated end-to-end acceptance test cases. In: Proceedings - 2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering, MiSE 2017, pp. 68–74 (2017)
17. Silva, A.R., Paiva, A.C.R., Silva, V.: 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) (2018)
18. Silva, A.R., Paiva, A.C.R., Silva, V.: A test specification language for information systems based on data entities, use cases and state machines. In: Hammoudi, S., Pires, L., Selic, B. (eds.) MODELSWARD 2018. CCIS, vol. 991, pp. 455–474. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-11030-7_20
19. 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. Verif. Reliab. **27**(3), e1629 (2017)
20. Moreira, R.M.L.M., Paiva, A.C.R.: 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 2014, pp. 863–866 (2014)
21. Paiva, A.C.R., Faria, J.C.P., Tillmann, N., Vidal, R.A.M.: A model-to-implementation mapping tool for automated model-based GUI testing. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 450–464. Springer, Heidelberg (2005). https://doi.org/10.1007/11576280_31
22. Silva, A.R.: Rigorous specification of use cases with the RSL language. In: Proceedings of International Conference on Information Systems Development 2019. AIS (2019)
23. Bhat, A., Quadri, S.M.K.: Equivalence class partitioning and boundary value analysis - a review. In: 2nd International Conference on Computing for Sustainable Global Development (INDIACom) (2015)
24. Paiva, A.C.R.: Automated specification-based testing of graphical user interfaces. Ph.D. thesis, Faculty of Engineering of the University of Porto, Porto, Portugal (2007)
25. ISTQB, ISTQB & #x00AE; Foundation Level Certified Model-Based Tester Syllabus (2015)
26. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Approaches and tools for automated end-to-end web testing. In: Advances in Computers, 1st edn., vol. 101, pp. 193–237. Elsevier Inc. (2016)
27. ISTQB, ISTQB & #x00AE; Foundation Level Extension Syllabus Agile Tester, p. 28 (2014)
28. Maciel, D., Paiva, A.C.R., Silva, A.R.: From requirements to automated acceptance tests of interactive apps: an integrated model-based testing approach. In: 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE) (2019)
29. Paiva, A.C.R., Flores, N.H., Faria, J.C.P., Marques, J.M.G.: End-to-end automatic business process validation. In: the 8th International Symposium on Frontiers in Ambient and Mobile Systems (FAMS) (2018)
30. Silva, A.R., Savic, D., et al.: A pattern language for use cases specification. In: Proceedings of EuroPLOP 2015. ACM (2015)
31. Maciel, D.A.M.: Model based testing - from requirements to tests. MSc thesis, Master in Informatics and Computing Engineering, Faculty of Engineering of the University of Porto, Portugal (2019)