Generating Test Cases from Use Cases and Structured Scenarios: Experiences with the RSL Language

Ana C. L. Gomes
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal *catarinag@gmail.com*

Ana C. R. Paiva
Faculdade de Engenharia da Universidade do Porto & INESC TEC
Porto, Portugal *apaiva@fe.up.pt*

Alberto Rodrigues da Silva
INESC-ID - Instituto Superior Técnico - Universidade de Lisboa
Lisboa, Portugal *alberto.silva@tecnico.ulisboa.pt*

Abstract

Requirements and tests written in natural languages are very popular, but also lead to ambiguities and misunderstandings. One way to diminish these problems is by using controlled natural languages, such as the RSL (Requirements Specification Language), that is used in this research. This paper proposes an approach that involves the generation of test suites and test cases from use cases and structured scenarios, all of them specified in that common language. This approach includes an additional task, in which users can manually refine and customize the generated test suites, consequently promoting automation and re-usability but also flexibility. The feasibility of the discussed approach is illustrated and discussed based on an e-commerce web site, properly designed for testing purposes.

Keywords: Requirements Specification, Test Case Specification, Use Cases, Use Case Scenarios, Test Cases Generation, RSL.

1. Introduction

The successful delivery of software products depends on how well the requirements have been understood and transformed into relevant product features [25]. A requirement can be defined as a condition or capacity that a system must have for satisfying the needs of its users [10].

A software requirements specification is a technical document that establishes an agreement between software developers and stakeholders, and shall show a clear understanding of the requirements in an unambiguous way taking into account two aspects [21] [14]: the need to make the requirements document readable and the need to make requirements document liable to processing. These concerns help avoid miscommunication, misinterpretation, and contribute to higher productivity, and time and cost savings [14].

The most common way to write requirements is with natural languages; however, this may also lead to ambiguities and inconsistencies. The literature shows that collecting ambiguous requirements is one of the critical challenges in software engineering [3]. To overcome this problem many researchers use formal languages complemented with models and semi-formal methods. The formal language has a well-defined syntax and semantics, which allows to define the elements and express possible relations between those elements [4]. However, the use of formal languages adds complexity and requires mathematical knowledge that can increase the costs [11]. Even when formal (like B or Z) and semi formal languages (like UML or SysML) are applied, there is always a need to use natural language sentences [16]. To get the best out

of both solutions (i.e., of natural languages and semi-formal languages), some authors have proposed controlled natural languages (CNLs) [17]. The usage of a CNL decreases the possibility of errors and misunderstandings during both the requirements and tests specification and may contribute to better validate these specifications. One example of a well-defined CNL for requirements engineering (RE) is RSL [6], which allows to specify requirements and tests in a rigorous way but still keeping the specifications human-readable. The support for the specification of both requirements and tests in an integrated way was the main reason for the adoption of RSL in our research. RSL is a CNL that allows requirements engineers and testers to specify requirements and tests in a systematic and consistent way [7]. RSL includes a set of constructs that represent different RE-specific concerns (e.g., active structure, behavior, passive structure, requirements or tests) and can be applied at different levels of abstraction (e.g., at business, application, software, or even hardware level) [7]. Furthermore, RSL supports the establishment of alignments between requirements and tests, allowing the generation of test cases from use case scenarios. This automation process brings benefits by reducing the effort and cost of repetitive tasks, and ensuring higher quality of both requirements and tests.

The following sections illustrate the specification of requirements and tests, as supported by the RSL language. To better support the explanation, we use a running example, which is a fictitious e-commerce website named "MyStore", particularly designed to support test automation practices[1]. The MyStore application includes features that allows its users to register, authenticate, create and update products, create orders of products and checkout, etc.

In a previous work, Maciel et al. [18] [22] discussed an "end-to-end approach" that intends to generate test scripts from requirements and tests specifications written manually in RSL. That approach (Figure 1) involves several tasks: Task-1 consists in the manual specification of requirements. Tasks-2a and -3a consist in the manual specification and refinement of tests. Task-4 involves the generation of tests scripts from the tests specifications with RSL; in the reported experience these test scripts were written in the Robot Framework's language. Task-5 is a manual and time-consuming activity that requires the mapping of the concrete GUI elements (of the system under test, SuT) with the elements defined in the test scripts. Finally, Task-6 consists in the tests execution against the SuT and produces a test report with the results.

In spite of being an ambitious and complex process, we identify some limitations with that approach. First, the Tasks-2a and -3a could be automated to some extent, and hence, could show a higher productivity. Second, the design of the RSL could be extended with more details, namely use case scenarios, and test cases with a sequence of concrete test steps. Consequently, as suggested in Figure 1, the key contributes of this work is to discuss how to extend and improve these new tasks of that end-to-end process [18] [22]: (Task-2a) Generate Test Suites; and (Task-3a) Refine Test Suites.

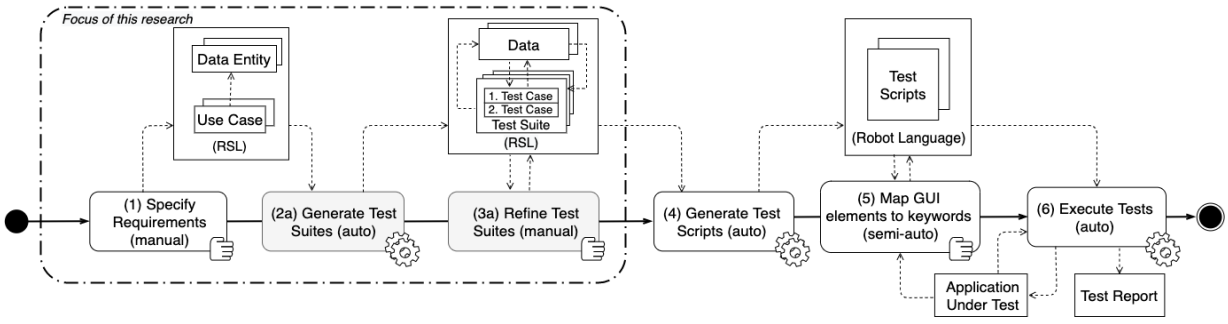

Fig. 1. General view of the proposed approach.

[1] http://www.automationpractice.com

This paper is structured as follows: Section 2 presents the RSL elements for the specification of requirements, namely for use cases and use case scenarios. Section 3 presents the RSL elements for the specification of tests, including the discussion of test suites, test cases, test steps and variables. Section 4 discusses the proposed approach in what regards the tasks Task-2a and Task-3a. Section 5 presents and discusses the related work; and Section 6 presents the conclusions and some open issues.

2. Requirements Specification

RSL allows the specification with different levels of abstraction and with different types, such as goals: functional requirements, constraints, use cases, user stories or quality requirements. This paper focuses on the RSL constructs particularly related with the specification of use cases, namely considering the following constructs: actor, use case, scenario, data entity and respective relationships [18] [22], that we briefly introduce below.

2.1. Data Entity

A data entity represents a structural entity that exists in information systems commonly associated with data concepts identified from the domain analysis. These entities can be represented by data structures, such as tables or collections that may include the specification of attributes, foreign keys and other check data constraints [6, 7]. Listing 1 shows a RSL specification of two data entities identified in the MyStore application: User and Product with their respective attributes, as well as data element (d_Users) with concrete data.

Listing 1. Specification of data entities and data elements (in RSL).

```
*** DataEntities ***
DataEntity e_User "User" : Master [
        attribute ID "ID" : Integer [primaryKey]
        attribute firstName "Name" : Text [isNotNull]
        attribute email "Email" : Text [isNotNull isUnique]
        attribute password "Password" : Text [isNotNull isEncrypted]
        attribute statusActive "StatusActive" : Boolean]
DataEntity e_Product "Product" : Master [
        attribute ID "ID" : Integer [primaryKey]
        attribute SKU "SKU" : Integer [isUnique]
        attribute Name "Name" : Text [isNotNull]]
*** Data ****
Data d_Users : e_User [
        withValues
        | e_User.ID | e_User.email      | e_User.password +|
        | 1001      | "john@email.com"  | "#password1234" +|
        | 1002      | "peter@email.com" | "#password1236" +| ]
```

2.2. Use case

A use case defines a set of behaviors of the system that gives an observable result of value for the related actors [5]. The behavior of a use case is generally described by one or more scenarios. A use case may have preconditions, which need to be met before its start and post-conditions, which defines the final state of the system after the use case is completed. A use case can be classified by a specific type such as: 'EntityCreate', 'EntityRead', 'EntityUpdate', 'EntityDelete' [6]. These use case types are commonly found in business information systems that support data management tasks [6] [8].

Listing 2. Specification of actors and a use case (in RSL).

```
Actor a_Admin "Administrator" : User
Actor a_Anonymous "Anonymous" : User
*** Use Case ***
UseCase uc_2_Login "Login" : EntityRead [ primaryActor a_Anonymous
                dataEntity e_User...]
```

2.3. Scenarios

A use case can be detailed by one or more scenarios, each one with a sequence of steps. A step is executed by an actor or by the system to fulfill the use case's goal. A use case has one main scenario and may have other alternative and exception scenarios.

Listing 3 shows the specification of the login use case's main scenario. In this scenario the system checks if the "username (e-mail)" and "password" provided by the user are recognized by the system. This specification includes two exception scenarios: (1) when the user is not registered, and (2) when the user fills in a wrong password. In both scenarios the system will report an error message.

Listing 3. Specification of a use case scenario (in RSL).

```
*** Use Case (scenarios) ***
mainScenario uc_2_Login_Main (Main) [
 step s1 (System:OpenBrowser) "Shows the login page"
 step s2 (Actor:PostData) "Fills in the email field" [
   scenario s2a_NotYetRegistered (Exception) "Not yet registered" [
     step s2a.1 (Actor:PostData) "Fills in an email field that is not
                 registered"
     step s2a.2 (System:Check) "Displays an error message" nextStep s2 ] ]
 step s3 (Actor:PostData) "Fills in the password field"
 step s4 (Actor:SubmitAction) "Clicks on the Login button"
 step s5 (System:Check) "Validates the pair <email, password>" [
   scenario s5a_WrongPasswordScenario (Exception) "Login with a
                 wrong password" [
     step s5a.1 (System:Check) "Displays an error message" nextStep s2 ] ]
 step s6 (System:ShowData) "MyAccount button appears with the user name" ]
```

3. Tests Specification

Figure 2 shows a partial view of the RSL metamodel with the key elements discussed in this paper. On the top of that figure it is shown that a UseCase has several Scenarios, a Scenario has several Steps. On the other hand, on the bottom, a TestSuite has several TestCases, and a TestCase has several TestSteps. In addition, a TestSuite (of type UseCaseTest) refers to a specific UseCase, and may use several concrete data elements (Data), which shall be conformed with a DataEntity. This section presents the main elements used to test specification, directly related with use cases, and also some advanced features (e.g., variables, setup and teardown actions) that support reusability and flexibility.

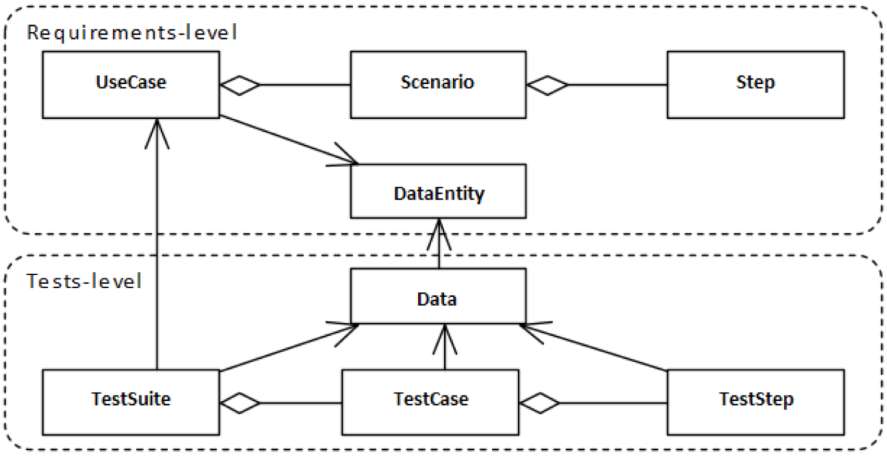

Fig. 2. Use case and Test Suite RSL elements (partial metamodel)

3.1. Test Suites

A test suite aggregates several test cases in which one corresponds to the main scenario and the other test cases to the alternative and exception paths. As suggested in Listing 4, a test suite includes an unique identifier (ID), a name, a type, a reference to a requirement (e.g. use case, user story, goal) and a set of test cases. The test suite's type constraints the type of the requirement assigned (this paper only concerns with test suites of type "UseCaseTest").

Listing 4. Specification of a test suite (in RSL).

```
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [ useCase ...]
```

3.2. Test Cases

Listing 5 shows a test suite with test cases, test steps, variables and references to data entities. Each test case has an unique identifier, a name that describes its purpose, and a set of test steps.

Listing 5. Specification of a test suite and test case (in RSL).

```
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [
 useCase uc_3_CreateProduct
 variable v_Products [ withData (d_Products)]

 testCase ts_3_1_CreateProduct "User Successfully Creates The Product" [
  step s1 (System:System_Execute) "System opens the browser in URL"
  step s2 (Actor:Actor_PrepareData readFrom v_Products.ID) "User fills in
                the Product  ID input field"
  step s3 (Actor:Actor_PrepareData readFrom v_Products.Name) "User fills in
                the Product Name input field"
  step s4 (Actor:Actor_CallSystem element('Create')) "User clicks on the
                'Create' button"
  step s5 (System:System_ReturnResult textOnElement(text vSuccess))
        "System verifies if the success message appeared" ]
```

3.3. Test Steps

A test step has an unique identifier, the identification of the entity who performs the action (e.g., System or Actor), an action, and a brief description. An action has the following possible types: Actor_PrepareData (actor's action to insert data), Actor_CallSystem (actor's actions, such as button click, checkbox select), System_Execute (actions executed by the system, e.g., 'OpenBrowser' and 'Check') and System_ReturnResult (to collect and store data in temporary variables).

A step is classified with an operation type and, eventually, an operation extension type. These operation types describe the actions that are performed in the respective step. The operation extension types are subtypes for the previous types that specify the action. Each step must have an operation extension that allows to specify the target element. Finally, the test check defines the validation performed in the step where it is specified.

3.4. Setup and Teardown Steps

Setup is used to set initial state before test suite execution. Teardown performs cleanup after test suite execution. A test suite execution runs all the inner test cases.

Listing 6. Specification of a test suite (in RSL).

```
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [ ...
 setup [ step setup_1 execute t_uc_1_SignUp.ts_1_1_Signup
                withVariable (v_Users) ]
*** Test Suite teardown ***
 teardown [
    step teardown_1 execute t_uc_6_DeleteUser.ts_6_1_DeleteUser
                withVariable (v_Users)
```

```
    step teardown_2 execute t_uc_5_DeleteProduct.ts_5_1_DeleteProduct
                withVariable (v_Products) ] ... ]
```

When a setup is defined, usually there is also a definition of a teardown to roll back the updates performed so as following test suite executions start from the same initial desired state.

Listing 7. Specification of a test case (in RSL).

```
*** Test Case setup ***
testCase ts_3_1_CreateProduct "User Successfully Creates The Product" [...
 setup [ step setup_1 execute t_uc_2_Login.ts_2_1_Login
                withVariable (v_Users[1]) ]...]
*** Test Case teardown ***
testCase ts_5_1_DeleteProduct "User Successfully Deletes The Product" [ ...
 teardown [ step teardown_1 execute t_uc_5_DeleteProduct.ts_5_1_DeleteProduct
                withVariable (v_Products) ] ... ]
```

3.5. Variables

Concrete data elements and variables are useful to specify concrete data that can be used by test suites and test cases. These elements can be defined at the level of a test suite (i.e., they can be accessed by all its test cases) or a test case (i.e., they only get accessed inside the test case).

Listing 8. Specification at test suite and test case levels (in RSL).

```
*** Test Suite ***
TestSuite t_uc_2_Login "Login" : UseCaseTest [
        useCase uc_2_Login
        variable vSuccess ["success"]
        variable vError ["error"] ... ]
*** Test Case Level ***
 testCase ts_3_1_CreateProduct "User Successfully Creates The Product" [
        variable v_Products withData (d_Products)
        step s1 (System:System_Execute) "System opens the browser in URL"
        step s2 (Actor:Actor_PrepareData readFrom v_Products[1].ID) "User
        fills in the Product ID input field" ... ] ]
```

Listing 8 shows the specification of a test suite with test suite level variables (i.e., vSuccess, vError). This group of variables can be later referred in test cases. Listing 8 also shows in the specification of a test case, the values assigned to the internal variables and inserted into the steps, which contain the necessary information to execute the flow of events.

It is possible to pass variables through parameters. Listing 9 shows two different ways to do that: (1) with a data element; or (2) with values.

Listing 9. Specification of parameters (in RSL).

```
*** Data ***
variable v_User is p_User otherwise withData (d_Users)
*** Values ***
variable v_User is p_User otherwise withValues (
   | ID             | firstName          | email           | password        +|
   | d_Users[1][0]  | d_Users[1][1]      | d_Users[1][2]   | d_Users[1][3]   +|)
```

Listing 10 shows the ts_2_1_Login test case that, when included by other tests, can receive variables by parameter values. The parameter variable is defined and assigned values from data or from local values in the test case. When the setup t_uc_3_CreateProduct test suite is executed, call the ts_2_1_Login test case and send the value through parameter. The user is able to login into the system and successfully create the product. The values assigned are defined in a tabular format with several rows.

Listing 10. Specification of variables (in RSL).

```
*** Login Test Suite ***
TestSuite t_uc_2_Login "Login" : UseCaseTest [
  useCase uc_2_Login
  variable v_Users [ withData (d_Users)]
```

```
  testCase ts_2_1_Login "User Logs In With Valid Information" [
        parameter p_User
        precondition v_Users.statusActive
        variable v_Users is p_User otherwise withData (d_Users) ....]
*** Create Product Test Suite ***
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [
  useCase uc_3_CreateProduct
  setup [step setup_1 execute t_uc_2_Login.ts_2_1_Login
                withVariable (v_Users)] ... ]
```

4. Proposed Approach

Due to the traditional separation between requirements and testing activities, it is not common to start the testing activity at the beginning of the software development process. Our approach tries to avoid this situation by supporting the specification of tests at an early stage of the project and also by generating such tests from requirements and ensuring their alignment.

The objective of our research is to define a process of generating and refining test suites from use cases defined in RSL. Our research also intends to extend the end-to-end approach previously discussed by Maciel et al. [18] [22].

The focus of our proposal (shown in the dashed rectangle in Figure 1) involves the following tasks: (1) specify requirements (2) generate test suites; and (3) manually refine test suites.

The MyStore running example is used to better support the explanation, considering the following requirements: (1) the user shall create an account on the MyStore (this account includes basic user information, such as username (e-mail) and password); (2) before using the application, the user shall authenticate to access the system; and (3) the user shall create and update the products (that will then become available for being sold).

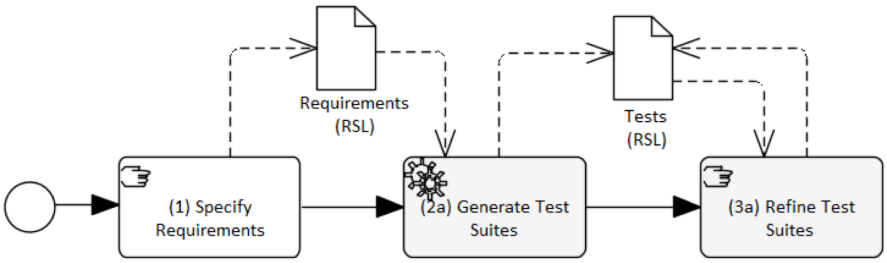

Fig. 3. Testing generation and refinement processes.

4.1. Specify requirements (Task-1)

Requirements are specified with RSL constructs, such as: data entity, data, actor, use case and respective relationships. In the running example we defined two data entities (User and Product), two actors (Administrator and Anonymous) and six use cases (sign up, login, create product, manage products, delete product and delete user). Some of these specifications are shown above in Listings 1 and 2.

4.2. Generate test suites (Task-2a)

RSL supports the specification of test suites and test cases directly from the requirements specifications.

The generation of test suites from a use case involves creating a test suite for each use case, and establishing a reference to the involved use case. Then, the use case scenarios (main, alternative and exception branches) are transformed into test cases. One test case is generated

for each scenario with respective steps and variables if relevant.

In the MyStore example the "User successfully creates the product", the test case steps are obtained from the respective use case main scenario. The test case has the steps to achieve the result and compare it with the expected value. User fills in valid product information following the steps and at the end the System verifies if the success message appeared.

Since, the manual specification of data is a repetitive and tedious process, the data elements are also generated with the test suites generation process, which reduces the time and costs of producing such tests.

Listing 11. Example of data specification (in RSL).

```
*** Data ***
Data d_Products : e_Product [
    withValues
        | e_Product.ID | e_Product.SKU    | e_Product.Name +|
        | 0001         | "9001"          | "Product 1"    +|
        | 0002         | "9002"          | ""             +| ]
```

4.3. Refine test suites (Task-3a)

The generated test specifications might be subject to manual refinements. After the transformation of the tests, the following activities can be performed: refinement, reuse and sequencing of the generated tests.

Data refinement

One of the first steps in refinement is to define concrete values to the data entities and create temporary variables. Listing 12 shows the "Create Product" test suite with the necessary refined information. This test suite has 4 variables. The first two variables define the success and error validation messages, and the last two variables define product and user data values.

Listing 12. Example of a test suite specification (in RSL).

```
*** Test Suite ***
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [
  useCase uc_3_CreateProduct

  variable vSuccess [ "success" ]
  variable vError [ "error" ]
  variable v_Products [ withData (d_Products)]
  variable v_Users [ withData (d_Users)] ... ]
*** Test Case ***
testCase ts_3_2_CreateProduct "User Does Not Fill In The Product ID And
                The System Auto-generated It" [
  variable v_Products withData (d_Products[2])

  step s1 (System:System_Execute) "System opens the browser in URL"
  step s2 (Actor:None  readFrom v_Products.ID) "User does not fill in the
                Product ID input field"
  step s3 (System:System_Execute writeTo v_Products.ID) "System auto-generates
                the Product ID"
  step s4 (Actor:Actor_PrepareData readFrom v_Products.Name ) "User fills
                in the Product Name input field"
  step s5 (Actor:Actor_CallSystem element('Create')) "User clicks on the
                Create button"
  step s6 (System:System_ReturnResult textOnElement(text vSuccess))
                  "System verifies if the success message appeared" ]
```

The test case contains the necessary information to perform the steps in order to achieve the results and compare it with the expected values. In Listing 12, the test case gets data from data entities defined. The tester chooses the actor/action types to apply and defines the operation extension values. Listing 12 shows the operation extension assigned values and respective data entity/variable attributes. To execute step 4, the actor prepares the data which reads from a respective data entity/variable attribute: v_Products.Name.

The test case ends with a check in order to evaluate the success/error at step 6 of the test. Whenever the tester needs it, it is possible to refine the data by editing or adding new records.

Reusing

Setup and teardown steps are two reuse mechanisms provided by the RSL. There are some advantages to use these reuse mechanism: First, easy to write and clear test specifications. Second, less chance of setting up too much or too little for the given test. And third, less chance of sharing state between tests, which creates unwanted dependencies between them.

Listing 13. Example of a test suite specification (in RSL).

```
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [ ...
 variable v_Products [ withData (d_Products)]
 variable v_Users [ withData  (d_Users)]

 testCase ts_3_1_CreateProduct "User Successfully Creates The Product" [
  setup [ step setup_1 execute t_uc_1_SignUp.ts_1_1_Signup
                withVariable ( v_Users ) ]
  step s1 (System:System_Execute) "System opens the browser in 'URL'"
  step s2 (Actor:Actor_PrepareData readFrom v_Products.ID) "User fills
                in the Product ID input field"
  step s3 (Actor:Actor_PrepareData readFrom v_Products.Name) "User fills in
                the Product Name input field"
  step s4 (Actor:Actor_CallSystem element('Create')) "User clicks on the
                'Create' button"
  step s5 (System:System_ReturnResult textOnElement(text vSuccess)) "System
                verifies if the success message appeared"]
  teardown [
   step teardown_1 execute t_uc_6_DeleteUser.ts_6_1_DeleteUser
                withVariable (v_Users)
   step teardown_2 execute t_uc_5_DeleteProduct.ts_5_1_DeleteProduct
                withVariable (v_Products)]]
```

Listing 13 shows the test suite "Create Product" with a "happy flow" test case. Since the sign up action is required for all test cases of the test suite, the tester manually defines a setup action with the test case that allows the user to sign up (i.e., the "SignUp" include action). Then, the sign up test case receives values by parameters which are defined in the test suite "SignUp" (Listing 14).

Listing 14. Example of a test suite specification (in RSL).

```
TestSuite t_uc_1_SignUp "SignUp" : UseCaseTest [
 useCase uc_1_SignUp
 variable v_Users [ withData (d_Users)]

 testCase ts_1_1_Signup "Anonymous User Signs Up With Valid Information" [
  parameter p_User
  variable v_Users is p_User otherwise withData ( d_Users ) ... ]
```

The parameter variable is defined and assigned values from a data entity in the test case. When the setup of the product creation test suite is executed, call the "Sign up" test case and send the values through parameters.

All the steps in the test case are performed successfully. The user is able to login into the system and successfully create the product. After running all the test cases, the teardown steps are executed and delete the records, which were created or changed during the test suite setup. The teardown mechanism shall clean up the data, even if some test cases do not pass, and all the data returns to the original state.

Sequencing

The tester can still manually define preconditions and postconditions. Preconditions and post-conditions allow to create sequencing mechanisms for the test cases. Precondition specifies the setup needed for a test case to be successfully executed following a specific sequence.

Listing 15. Example of a test case precondition specification (in RSL).

```
TestSuite t_uc_1_SignUp "SignUp" : UseCaseTest [
  useCase uc_1_SignUp
  setup [ step setup_1 execute t_uc_6_DeleteUser.ts_6_1_DeleteUser
            withVariable (v_Users)
  ]
  testCase ts_1_1_Signup "Anonymous User Signs Up With Valid Information" [
    parameter p_User
    precondition v_Users "statusActive" ...] ... ]
```

In the MyStore example, the "Sign Up" test case starts when an actor registers into the e-commerce system with valid information, having the following precondition: “The user does not exist in the system”. Consequently, the precondition must be true before the test case is allowed to proceed. The postcondition for a test case shall also be true regardless of the flow being executed. Listing 16 shows the test case "Create Product" with the following postcondition: the product ID (v_Products[1].ID) must be created in the system.

Listing 16. Example of a test case postcondition specification (in RSL).

```
*** Test Suite ***
TestSuite t_uc_3_CreateProduct "Create a Product" : UseCaseTest [
  useCase uc_3_CreateProduct
  postcondition v_Products[1].ID ... ]
```

5. Related work

Requirements and tests written in natural language are easy to understand as no special technical knowledge is required. However, the use of natural language raises some challenges because it may be ambiguous [9], inconsistent, incomplete, and incorrect [23]. There are some proposals or requirements specification techniques that range from formal, textual or graphical.

The representation of requirements influences the techniques used to write or generate test cases. Some approaches have used natural language processing (NLP) techniques to parse, extract, and analyze sentences and have contributed to reduce the quality problems of these sentences, like ambiguity or inconsistency. However, researchers on NPL have focused more on identifying ambiguity than trying to avoid or resolve it [2]. The ambiguity must be addressed at an early stage by representing requirements in a machine-readable format [26].

Other techniques generate test cases from UML use case diagrams. Wang et al. [27] discuss how to generate executable acceptance tests by exploiting behavioral information in use case specifications. They propose a method, called Restricted Use Case Modeling (RUCM), that introduces a template with keywords and restriction rules to reduce ambiguity in requirements. That includes an advanced NLP solution to generate Object Constraint Language (OCL) constraints that capture pre and postconditions of use case steps. The Use Case Test Models (UCTM) captures the control flow implicitly described in a RUCM specification and enables the model-based identification of use case scenarios which significantly reduces time and manual human effort.

Hamza et al. [13] discuss how to convert UML use case diagrams into customized activity diagrams. Then, these activity diagrams are converted into directed activity graphs (DAGs), from which test cases are extracted. However, this technique does not deal properly with the order in which test cases should be executed because of existing dependencies among them.

Alrawashed et al. [1] and Somé et al. [24] apply Cockburn’s template [5] to specify use cases using a structured natural language. Similarly to our approach, the use case steps are represented by nodes, and edges represent the sequence among them. Then, the steps are converted to create a control flow. After creating the control flow, the two approaches diverge. Alrawashed et al. [1] generate test cases from the control flow diagram with a genetic algorithm to optimize and evaluate the adequacy of generated test cases. On the other hand, Somé et al. [24] generates test cases using depth-first traversal of the control flow but in a not completely automated man-

ner. Our approach goes beyond these two approaches by generating test cases and data using information defined in the specifications.

Fischbach et al. [12] translate acceptance criteria into a Cause-Effect-Graph (CEG) to automatically derive the test cases. First, each acceptance criterion is converted into a dependency tree. Then, the algorithm traverses such tree to extract the causes and effects. Finally, the algorithm derives the minimum number of test cases from the CEG by applying the Basic Path Sensitization Technique (BPST) [12]. However, the full generation of test cases, from acceptance criteria, can hardly be achieved and, so, according to some practitioners, this approach should be seen as a supplement to the existing manual process [12].

In a model based testing (MBT) approach, test cases may be generated from models. These models may have a textual [15] or graphical representation [6] and are a more structured way to specify requirements and tests. One example of such an approach is Paiva et al. [20].

Some researchers suggest the use of formal languages complemented with models and semi-formal methods [19]. However, the use of formal languages increases the complexity and requires some mathematical knowledge that also increases cost [11].

Some researchers have proposed CNL as extensively discussed by Kuhn [17]. These languages, such as RSL, benefit from being closed to natural language and also provide some structure that allow automatic manipulation, decreasing the possibility of errors and misunderstandings during both the requirements and tests specification. In [18] [22], acceptance tests are generated from RSL specifications. This paper extends this work by increasing the automation degree of the process. Maciel et al. [18] [22] discuss a preliminary end-to-end approach to generate test scripts from requirements and tests specifications written manually in RSL. However, they do not propose how to automatize their Task-2a (Specify Tests) from use cases. The proposal discussed in this paper was focused on this issue, and we show how (1) to extend the RSL language; and (2) to implement generation algorithms to achieve this purpose.

6. Conclusion

This paper presents an approach to automate the generation of test suites and test cases from use cases and structured scenarios, both specifications defined in RSL. This paper also introduces advanced mechanisms – such as data variables, setup and teardown blocks of actions –, not commonly found in the related work, which allow manually refining the test cases in a flexible and reusable way. The discussion of the proposed approach is illustrated with a running example, the MyStore e-commerce application.

For the future work, we aim to integrate these contributions with the test automation process, as originally proposed by Maciel et al. [18] [22], namely, with test automation and robot process automation tools. We also plan to extend this research to support other types of requirements and tests, e.g., user stories and cyber-security testing. We intend to explore machine learning and artificial intelligence techniques to improve the generation of concrete test data. Finally, we aim to apply and validate this research with more case studies.

Acknowledgements

Research partially funded by FCT UIDB/50021/2020 and LISBOA-01-0145-FEDER-029360 e PTDC/CTA-OHR/29360/2017.

References

1. Alrawashed, T. A., Almomani, A., Althunibat, A., Tamimi, A.: An Automated Approach to Generate Test Cases From Use Case Description Model. In: CMES-Computer Modeling in Engineering & Sciences, 119(3) (2019)
2. Bano, M.: Addressing the challenges of requirements ambiguity: A review of empirical litera-

ture. In: Fifth Int. Workshop on Empirical Requirements Engineering (EmpiRE). IEEE (2015)
3. Besrour, S., Rahim, L. B. A., Dominic, P. D. D.: A quantitative study to identify critical requirement engineering challenges in the context of small and medium software enterprise. In: 3rd Int. Conference on Computer and Information Sciences (2016)
4. Bork, D., Fill, H.: Formal Aspects of Enterprise Modeling Methods: A Comparison Framework. In: 47th Hawaii International Conference on System Sciences (2014)
5. Cockburn, A.: Writing Effective Use Cases, the Crystal Collection for Software Professionals. Addison-Wesley (2000)
6. Da Silva, A. R.: Rigorous specification of use cases with the RSL language. In: ISD 2019. AIS (2019)
7. Da Silva, A. R., Paiva, A. C. R., da Silva, V. E.: A Test Specification Language for Information Systems Based on Data Entities, Use Cases and State Machines. In: Int. Conference on Model-Driven Engineering and Software Development (2018)
8. Da 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)
9. Dalpiaz, F., Ferrari, A., Franch, X., Palomares, C.: Natural language processing for requirements engineering: The best is yet to come. In: IEEE Softw., vol. 35, no 5, p. 115–119 (2018)
10. Dube, R. R., Dixit, S. K.: Process-oriented complete requirement engineering cycle for generic projects. In: ICWET 2010. ACM (2010)
11. Ferreira, D., da Silva, A. R.: RSL-IL: An interlingua for formally documenting requirements. In: 3rd Int. Workshop on Model-Driven Requirements Engineering (MoDRE). IEEE (2013)
12. Fischbach, J., Vogelsang, A., Spies, D., Wehrle, A., Junker, M., Freudenstein, D.: SPECMATE: Automated Creation of Test Cases from Acceptance Criteria. In: IEEE (2020)
13. Hamza, Z. A., Hammad, M.: Analyzing UML use cases to generate test sequences. In: International Journal of Computing and Digital Systems 10 (2021)
14. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer, London (2011)
15. Júnior, V. A. de S., Vijaykumar, N. L.: Generating model-based test cases from natural language requirements for space application software. In: Software Quality Journal 20 (2012)
16. Kamsties, E.: Understanding ambiguity in requirements engineering (2005)
17. Kuhn, T.: A survey and classification of controlled natural languages, Computational Linguistics, vol. 40, no. 1, pp. 121–170 (2014)
18. Maciel, D., Paiva, A. C. R., da Silva, A. R.: From Requirements to Automated Acceptance Tests of Interactive Apps: An Integrated Model-based Testing Approach. In: Proceedings of ENASE'2019. SCITEPRESS (2019)
19. Mandrioli, D., Morasca, S., Morzenti, A.: Generating Test Cases for Real-Time Systems from Logic Specifications. In: ACM Transactions on Computer Systems (TOCS), 13, (n. 4). ACM (1995)
20. 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. In: Softw. Test. Verification Reliab. 27(3) (2017)
21. Nigam, A., Arya, N., Nigam, B., Jain, D.: Tool for Automatic Discovery of Ambiguity in Requirements. In: Int. Journal of Computer Science Issues, vol. 9, no. 5, pp. 350–356 (2012)
22. Paiva, A. C. R., Maciel, D., da Silva, A. R.: From Requirements to Automated Acceptance Tests with the RSL Language. In: Communications in Computer and Information Science, vol 1172. Springer (2020)
23. Raharjana, I. K., Siahaan, D., Fatichah, C.: User Stories and Natural Language Processing: A Systematic Literature Review. In: IEEE Access, vol. 9, pp. 53811-53826 (2021)
24. Somé, S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: SAC '08. ACER (2008)
25. Shah, U. S., Jinwala, D.: Resolving ambiguities in natural language software requirements: A comprehensive survey. In: ACM SIGSOFT Softw. Eng. Notes, vol. 40, pp. 1–7. ACM (2015)
26. Umber, A., Bajwa, I. S.: Minimizing ambiguity in natural language software requirements specification. In: 2011 Sixth Int. Conference on Digital Information Management (2011)
27. Wang, C., Pastore, F., Goknil, A., Briand, L.: Automatic Generation of Acceptance Test Cases from Use Case Specifications: an NLP-based Approach. In: arXiv: Software Engineering (2019)