Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 1569–1574
DOI: 10.15439/2014F342
ACSIS, Vol. 2

Automating Acceptance Testing with tool support

Tomasz Straszak, Michał Śmiałek
Warsaw University of Technology
Warsaw, Poland
Email: {straszat, smialek}@iem.pw.edu.pl

***Abstract*—During acceptance testing different areas of delivered software system are reviewed. Usually these are functionality, business domain logic, non-functional characteristics, user interface. Although they are related to the same particular functional area, they are verified separately. This paper presents the concept and the Requirements Driven Software Testing (ReDSeT) tool, which allows for automatic integrated test generation based on different types of requirements. Tests are expressed in newly introduced Test Specification Language (TSL). The basis for functional test generation are detailed use case models. Furthermore, by combining different types of requirements, relations between tests are created. The constructed tool acknowledges validity of the presented concept.**

I. Introduction

SOFTWARE testing is one of the main steps of each development process. In this step the compliance of delivered software with the requirements is being verified. Verification procedures of comparing the system under development to its requirements and needs of its users are encapsulated in the form of acceptance tests [1]. These requirements should be understandable for the stakeholders and at the same time precise enough for the developers to produce efficient software.

To describe the expected functionality of the software system, use cases are commonly used [2]. Use cases describe interactions between external actors and the system, which lead to specific goals according to the given scenarios. Such requirements can be claimed as satisfactory to define the tests, that will be performed during acceptance testing.

A number of automatic test generation mechanisms based on use cases were proposed. Examples of such approaches can be found in work by El-Attar and Miller [3], Gutiérrez et al. [4], and Nebut et al. [5]. Beside use cases, requirement specifications contain other types of requirements, that describe different aspects of the desired software. These requirements should also be verified by executing corresponding tests. Some work has been done on the generation of tests based on business rules (see Junior et al. [6]), GUI requirements (see Bertolini and Mota [7]), and even on non-functional requirements (see Dyrkom and Wathne [8]).

All these mechanisms use model transformation, forming the area of Model-based testing (MBT), which is an evolving technique for generating suites of test cases from requirements [9]. Although different types of tests are generated from requirements models describing the same software system, usually they are not related, because they verify different aspects of the system.

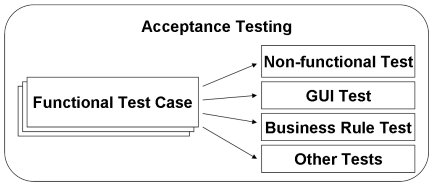

Fig. 1. Acceptance test suite based on functional test cases

This article extends the basic description of the idea presented in our previous work [10]. It focuses on automatic generation of different types of tests, integrated in functional test cases with test scenarios executed during acceptance testing. These tests are generated on the basis of interrelated requirements describing many aspects of the developed software system, making this idea MBT-compliant. The element that joins the different types of testing is the functional test case related to the use case scenario as shown in Figure 1.

This concept is based on the test metamodel defined as the Test Specification Language (TSL) and implemented within the ReDSeT tool (Requirements Driven Software Testing). The tests are generated automatically based on the requirement specification created with the Requirements Specification Language (RSL) [11]. As RSL provides notation for precise use case scenarios, generation of test cases verifying the system behaviour is significantly facilitated. Additional information contained in scenario sentences (notions from the domain vocabulary) and other related requirements allow for generating tests of different types. All the tests generated on the basis of RSL-based requirements form a complete test suite for acceptance testing.

II. Detailed requirements expressed in RSL

As in other test generation solutions, the basis for automatic generation of tests is the precise specification of requirements. As mentioned above, the described solution is based on requirements specifications created with RSL. The main features of this language are: clear separation of descriptions of the system's behaviour and descriptions of the system's domain. Functional requirements can be presented in three equivalent forms: structured text with hyperlinks to domain elements, an activity diagram or a sequence diagram. The elements describing the system's domain are depicted as notions on so-called notion diagrams. Each notion has operations that can be performed in regard to the particular notion. RSL allows for precise specification of requirements, which is understandable even for ordinary people who do not have technical expertise.

978-83-60810-58-3/$25.00

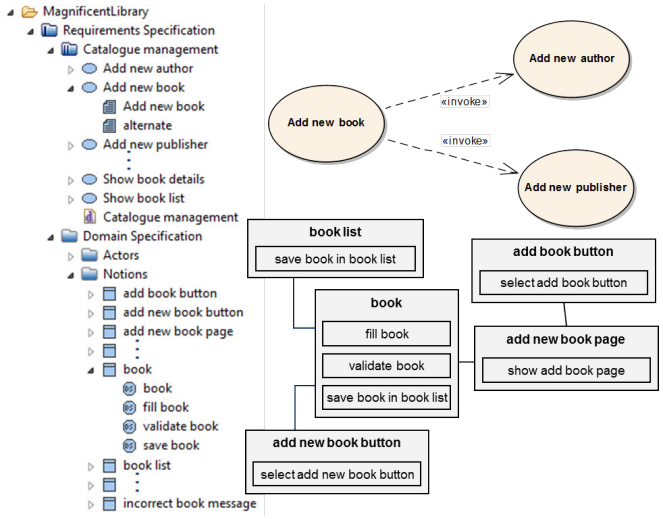

Fig. 2. Example of requirement specification structure, use cases and notion diagrams expressed with RSL

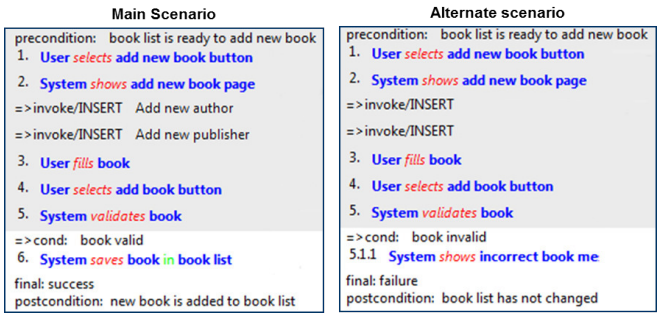

Fig. 3. Use case scenarios - textual representation

The language has a precise specification of its syntax and semantics [11] with methods of its use explained e.g. by Nowakowski et al. [12]. Figures 2, 3 and 4 shows an example requirements specification, created in RSL.

All the elements of a requirement specification are grouped in packages in a tree structure. Simple requirements described with free text can be used to define business rules or non-functional aspects of the system. Use cases defining the functionality of the system are described with structured scenarios (see Figure 3). Scenarios consist of numbered sentences in a simple grammar called SVO-O (Subject Verb Object - indirect Object). These sentences are constructed with notions stored in the domain vocabulary. This is illustrated with two scenarios (main and alternative) of the *Add new book* use case. The same information is presented in Figure 4 in the form of an activity diagram that is generated automatically from the scenarios.

The notions are referred-to in scenario sentences through hyperlinks (*book*, *book list*, *edit book button*, *edit book page*) and are presented on a notion diagram, that is similar to a class diagram. The relationships between notions, and notion operations are defined automatically according to the scenario sentences where these notions appear or are defined manually by the requirements engineer. The notions and their operations used in use case scenarios, describe the business logic and the user interface elements.

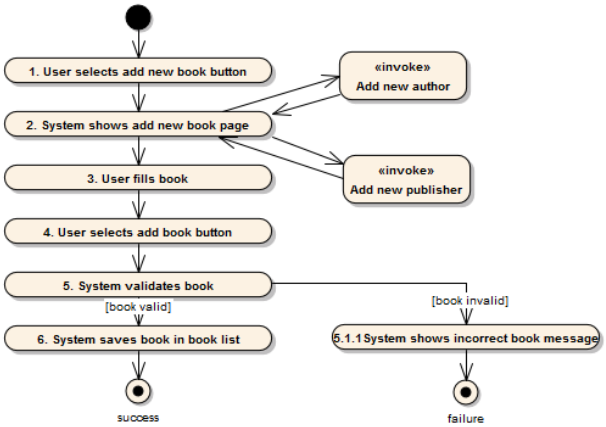

Fig. 4. Use case scenarios - activity representation

All the requirements can be related. To depict relations between use cases, a special invoke relationship is used (see Figures 2 and 4). It allows to determine under what conditions and in which step of a use case scenario another use case is to be called (see Nowakowski et al. [12] for more details). What is important, RSL is based on a formal metamodel. This allows for automatic processing of information contained in the requirement specification. This characteristics of RSL will be used for generating test cases.

III. Automating Test Generation

To define acceptance test suite and to ensure accurate and automatic transition from RSL-based requirements to tests, Test Specification Language (TSL) was developed. This language is based on a metamodel defined in the Eclipse Modeling Framework (EMF) [13] and is out of scope of this paper.

The main idea of TSL is to provide notation for reusable tests, that are understandable for non technical people and precise enough for detailed verification of the software system. All tests are grouped in a tree structure, called the Test Specification (see Figure 5), that groups tests assigned to a specific release of software. Each test contained in a test specification represents a procedure for verifying software in the context of a single requirement. Such a verification is made by examining all the check points defined inside the test.

The basic structure of a TSL test specification consists of two packages: Abstract Tests and Concrete Tests. The first of these includes tests generated directly from the requirement specification: mostly use case tests, notions, as well as tests of other types. A use case test corresponds to a use case, and includes test scenarios, as shown in Figure 5. Tests of other types, in addition to use case tests (verifying the behaviour of the system), can verify the business logic, user interface, non-functional aspects (performance, usability, etc.) or any other aspect of the system that is described through requirements.

A use case test scenario includes the initial condition (a precondition sentence) that must be met before the execution

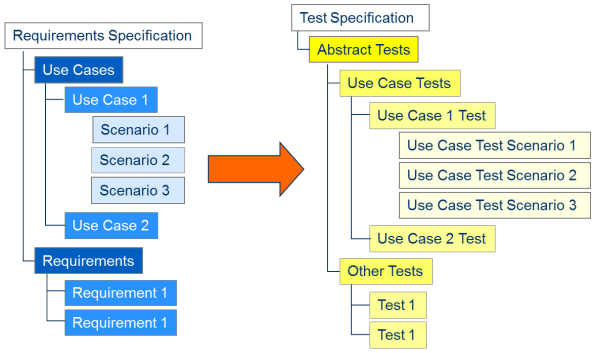

Fig. 5. Test generation based on the requirement specification

of actions described in this scenario and the final condition (a postcondition sentence) that describes the desired state of the system after the scenario is executed.

Every use case test scenario, generated from an RSL use case scenario, is a sequence of actions forming a dialogue between the primary actor and the system. Every such action is expressed by a single sentence in the SVO grammar (see Graham [14] for an original idea). These sentences describing single actions can have check points assigned. In addition to action sentences, two additional sentence types were introduced: condition and control sentences. They are used in a scenario to express the flow of control between alternative scenarios of the same use case as well as between scenarios of different use cases (see work by Śmiałek et al. [15]).

An important feature of requirement specified with RSL, is the possibility to create relationships. Due to generation of test specifications on the basis of these requirements, also relationships between tests can be created. The invocation relations between use cases are translated to become relations between use case tests. This provides information on the steps of the use case test scenario and on the conditions under which scenarios of other use case tests should be called. Relationships to other requirements are translated to relationships from use case tests to tests of other types.

Having two languages (RSL and TSL), which have definitions that are based on metamodels, automatic transformation from requirements to tests becomes possible also allowing for further acceptance test composition. There is a couple of common rules applied in the transformation:

- The structure of the packages containing use cases and notions is reflected in the structure of the packages in the Test Specification.
- Each element of a Test Specification that reflects an element of a requirement specification holds the identifier and the tree path of that element.

The transformation is performed in several steps according to the transformation procedure, that is presented in Figure 6. At the beginning of the transformation a new Test Specification structure is created (step 1). The basic Test Specification structure consists of a root node named using the pattern of "Software Case Name - date" and two child Test Packages named "Abstract Tests" and "Concrete Tests".

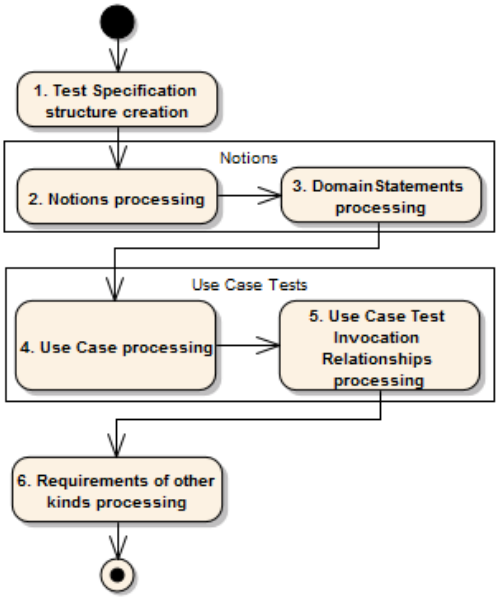

Fig. 6. RSL to TSL transformation procedure

All SVO sentences within the use case test scenarios are built of the notion's domain statements. For this reason, RSL notions should be transformed first (step 2). For each RSL notion, a TSL notion is created. The name, description and attached notion attributes are transferred.

For all TSL notions, domain statements contained in corresponding RSL notions are created (step 3). For each RSL notion, its domain statements are transferred into a TSL domain statement. The phrases contained in the RSL domain statement notions, used as direct and indirect objects, are pointed to by the *directNotion* and the *indirectNotion* attributes.

Having the notions with the domain statements transferred, the use case tests can be processed (step 4). For each RSL use case, a use case test is created and placed in a proper test package within the use case test structure. The name and the description are transferred directly. All the scenarios contained by the RSL use case are transferred into a use case test scenario. On the basis of the RSL scenario pre- and postcondition, adequate pre- and postonditions are created and attached to the use case test scenario. The sentences of the RSL scenarios are transferred into the correct specialisation of the use case test scenario sentence. The ordering number and the sentence text are set. For every SVO sentence, a proper domain statement is found and a relation to the corresponding domain statement is created. For every control sentence, a test invocation relationship is created with an empty use case test as its target.

The target use case tests of the test invocations are set after all the use cases are transferred into the use case tests (step 5). For each test invocation relationship contained in the control sentences, a correct use case test is found and set.

At the end of the transformation (step 6), tests of other kinds are created. Each RSL requirement that is not a use

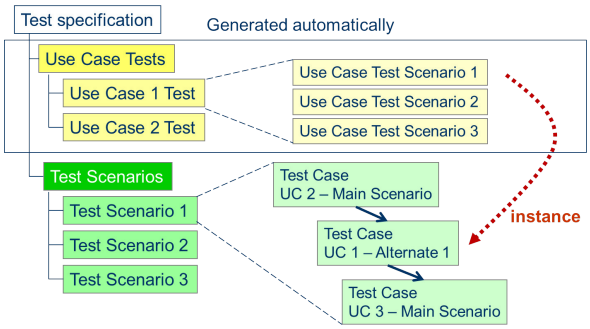

Fig. 7. Test scenarios composed of test cases

case and is classified as a requirement of specific type (e.g. business logic requirement, user interface requirement, non-functional requirement) is the basis for generating a test, which supplements use case tests, scenarios, sentences or notions. RSL requirements relations between use cases, notions and requirements of specific types are transformed into test relationships. As RSL currently does not support basic requirements of specific types, only non-functional requirements are automatically transformed into non-functional tests.

IV. Instantiating Tests

A scenario of a use case test determines the conditions, steps and check points that will be subject to verification for the use case implementation. These elements will be used in acceptance testing after placing them in test scenarios and assigning specific test data values. Test scenarios are grouped within second-level packages in the basic structure of a test specification named Concrete Tests, as shown in Figure 7. They are defined by a test engineer as a set of ordered instances of use case test scenarios, that we call test cases. A test case describes a procedure for verification of a system's functionality and is composed of ordered steps in the form of SVO sentences. Each step can contain check points with assigned test data values and can be related with instances of other type tests. These other test instances are automatically created during instantiation. They are related to a test case, just as abstract tests of other types are related to use case test scenarios and particular scenario sentences.

A test scenario constructed with test cases also builds the context for the test data. The initial test data values are set by the test engineer as the precondition values of the test scenario. Test data values describe basic business objects as well as GUI elements. The test data in the scope of one test scenario is passed between test cases as their pre- and postcondition values. The test data values are changing according to the functionality and the business logic that is under tests. It can be noted that although the test cases cannot be formally related to each other, within the manually created test scenarios, they indirectly refer to business processes that are implemented within the system being tested.

The instantiation procedure of a use case test scenario is performed in at least as much steps as the test scenario is supposed to have. Figure 8 presents the procedure for

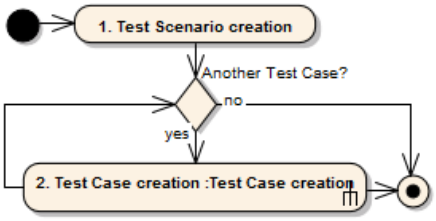

Fig. 8. Test Scenario creation procedure

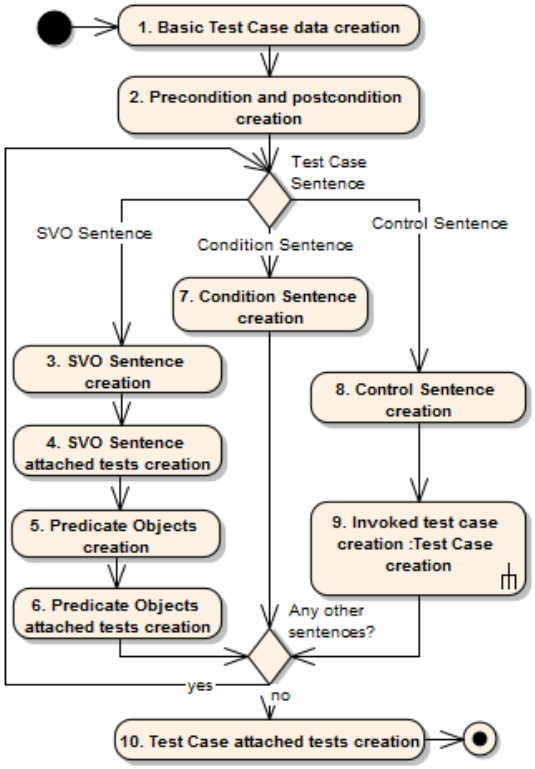

Fig. 9. Test Case instantiation procedure

test scenario creation. The test scenarios are inserted in the instance tests package. During creation, the name and the description of the test scenario should be given. All the steps of a test scenario are created as test cases. The number of test cases depends on the test engineer, who defines the steps of the test scenario. Each time a new test case is created, the instantiation procedure is performed. The procedure is presented in Figure 9.

At the beginning of the procedure, the test case order number is set. For each nested test case, the order number is segmented, e.g.: *2.3.1*. The name of the chosen use case test scenario and the description of the corresponding use case test are transferred into the test case (step 1), the same as for the use case test scenario pre- and postcondition (step 2).

Having the test case created, SVO, condition and invoke sentences are being created. SVO sentences (step 3) are transferred with their sentence order number and sentence text. Direct and indirect objects of the sentence predicates are

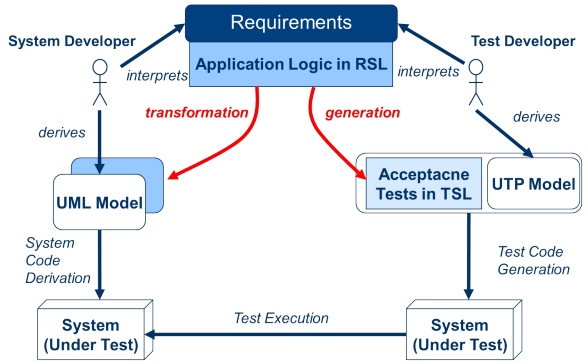

Fig. 10. TSL as UML Testing Profile complement

created on the basis of the direct and the indirect notions of the domain statement (step 5). All abstract tests of other types, related to the domain statement pointed to by the predicate relation and by the domain statements' direct notion and indirect notion relations, are transferred into adequate instance test specialisations (step 4 and 6). These instance tests of other types are contained in the SVO sentences of the test case and in the direct or the indirect object. The condition sentences are transferred with the sentence order number and the sentence text (step 7).

Depending on the test engineer's decision, the nested test case depicted by the control sentence can be created on the basis of a use case test related through a test invocation relationship to the currently processed use case test (step 8 and 9). If the use case test invocation is used, one of the invoked use case test scenarios in instantiated recursively. At the end all other abstract tests related to a use case test scenario are transferred into adequate instance test specialisations (step 10). These tests are contained within the test case.

V. TSL as an Extension of the UML Testing Profile

TSL can be seen as a stand-alone language but it can be easily interfaced with other languages for model-based testing. Prominently, it can be used in conjunction with the UML Testing Profile [16].

Figure 10 presents an appropriate usage scenario. Starting from requirements, a system developer delivers UML models, which are the basis for developing the system. The same requirements are used for manual creation of UTP models. On the basis of these test models, detailed unit and integration tests can be executed. However, this does not include high-level acceptance tests. Here, TSL offers an extension allowing to derive such tests directly and automatically from functional requirements.

The usage of RSL for defining application logic allows for automatic transformation of requirements into code and into acceptance tests in TSL. During these transformations, requirements-to-UML models and requirements-to-TSL-test traces are being created. These traces facilitate linking of UTP models with corresponding acceptance tests in TSL through UML models.

VI. Tool Support

The tool supporting the described idea of automatic test generation based on requirements is called ReDSeT (Requirements Driven Software Testing). It is based on the Eclipse Rich Client Platform. This enables integration with the ReDSeeDS tool (www.redseeds.eu, [17]) which provides advanced editors for requirements (for use cases, notions and other requirements) described with RSL. The generated test specification can be included in the same Eclipse project as the requirements specification and code. This allows for integration of activities at different stages of the software development project. As the repository of the test specification is based on the EMF technology [13], the TSL meta-model can be easily extended in order to handle other types of tests that are adapted to different types of requirements associated with use cases.

To start working with the ReDSeT tool, the requirement specification should be transformed into the test specification. When the automatic transformation is complete, the test engineer is able to manage the test specification organised in a tree structure using the Test Specification Browser and other dedicated editors enclosed in the ReDSeT perspective. Use case tests and use case test scenarios, along with test scenarios and test cases are presented in the Test Editor area. The Detailed Test View is dedicated for viewing check points and editing test values contained in all the types of tests. To create test scenarios and to instantiate use case test scenarios as test cases, dedicated wizards are available.

In order to perform acceptance tests according to test scenarios defined in the ReDSeT tool, the test execution scripts need to be generated. The test scripts, that are composed of test cases, contain detailed steps for the testers in the form of structured text. Each line represents one test with its name, description, input data values and expected state of the system for the specified elements.

The example of test execution script in the form of a CSV text file is presented in figure 11. The rows describing steps of a test scenario have solid background and are shown as numbered test cases. Each sentence of the test case is numbered with the test case number and the SVO sentence number. For each direct and indirect object of the SVO sentence, an additional row for data input or output appears. For example, in the step *2.1 SVO 3*, there are presented the input test data and the values for the author attribute (row number *2.1.SVO 3 DirObj*). Additional tests of other types, related to a sentence or to a direct or indirect object appear as separate rows. For example, a GUI test is presented in the step *2 SVO 2 DirObjGui Test*. This test is attached to the direct object of the step *2 SVO 2*.

On the basis of such a test execution script, the testers can verify the delivered software system. The results of each test step can be noted in an additional column. In case the test fails, the corresponding requirement can be found by tracing to the appropriate requirements element. The implementation units can be precisely located by examining traces to code, as described by Śmiałek et al. [18].

Test Type	Step	Test Name	Description	Test Data	Input Value	Characteristic to test	Expected result
Test Case	1	Show book list	---	---	---	---	---
SVO Sentence	1 SVO 1	User selects show book list button	---	---	---	---	---
Domain Object	1 SVO 1 DirObj	show book list button	---	---	---	---	---
SVO Sentence	1 SVO 2	System fetches book list	---	---	---	---	---
Domain Object	1 SVO 2 DirObj	book list	---	---	---	---	---
SVO Sentence	1 SVO 3	System shows book list page	---	---	---	---	---
Domain Object	1 SVO 3 DirObj	book list page	---	---	---	---	---
Test Case	2	Add new book	---	---	---	---	---
SVO Sentence	2 SVO 1	User selects add new book button	---	---	---	---	---
Domain Object	2 SVO 1 DirObj	add new book button	---	---	---	---	---
SVO Sentence	2 SVO 2	System shows add new book page	---	---	---	---	---
Domain Object	2 SVO 2 DirObj	add new book page	---	---	---	---	---
GUI Test	2 SVO 2 DirObjGUI Test	Asterisk marking mandatory fields	Empty mandatory fields should be marked with asterisk	---	---	Asterisk marking mandatory fields, when they are set	Empt mandatory fields marked with asterisk
Control Sentence	2 Control	INSERT Add new author	---	---	---	---	---
Test Case	2.1	Add new author	---	---	---	---	---
SVO Sentence	2.1 SVO 1	User selects add new author button	---	---	---	---	---
Domain Object	2.1 SVO 1 DirObj	add new author button	---	---	---	---	---
SVO Sentence	2.1 SVO 2	System shows add new author page	---	---	---	---	---
Domain Object	2.1 SVO 2 DirObj	add new author page	---	---	---	---	---
SVO Sentence	2.1 SVO 3	User fills author	---	---	---	---	---
Domain Object	2.1 SVO 3 DirObj	author	---	Author's name and surname	Mark Twain	---	---
SVO Sentence	2.1 SVO 4	User selects add author	---	---	---	---	---

Fig. 11. Test execution script - attached test and SVO sentence object test

VII. Conclusion

The proposed idea and the ReDSeT tool bring a complete solution for creating acceptance tests for the systems that are focused on user-system interaction. The preparation of test specifications can begin during the requirements formulation stage. Consecutive generation of tests allows to reach test complexity that corresponds to the level of detail of the final requirements. The basis for creation of a set of test scenarios are detailed use cases. Requirements defined in RSL significantly facilitate automatic test generation, and TSL allows for expressing interrelated tests in a way that is comprehensible to the audience responsible for acceptance testing. It can be noted that the proposed method is based on black box testing and is independent of the implementation technology of the system under test. On the other hand, since RSL and TSL are based on metamodels, the whole idea is close to Model Based Testing which goes into the details of system design.

In terms of future work, traces from requirements to test cases are planned to be used for generating requirements with test coverage reports. These traces will be subject to further research on regression test selection. Another area that is planned to be a subject of further research is using RSL, TSL and model transformations [19] as implementation of Test Driven Development (TDD) [20] and Behaviour Driven Development (BDD) [21]. These ideas assume that test effort is already incorporated at an earlier point of software development process. The proposed solution seems to have potential for automating the process of generating unit and acceptance tests and executing them on the basis of RSL requirements. It is also planned to develop the mechanism for extracting of test scripts as input for tools that automate test execution (e.g. IBM Rational Functional Tester, Selenium). It would bring a complete solution for detailed use case based testing.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley Publishing, 2011.

[2] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2000.

[3] M. El-Attar and J. Miller, "Developing comprehensive acceptance tests from use cases and robustness diagrams," *Requir. Eng.*, vol. 15, no. 3, pp. 285–306, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00766-009-0088-6

[4] J. J. Gutiérrez, M. J. Escalona, M. Mejías, and J. Torres, "An approach to generate test cases from use cases," in *Proceedings of the 6th international conference on Web engineering*, ser. ICWE '06. New York, NY, USA: ACM, 2006, pp. 113–114. [Online]. Available: http://doi.acm.org/10.1145/1145581.1145606

[5] C. Nebut, F. Fleurey, Y. L. Traon, and J. marc Jézéquel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 140–155, 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.22

[6] E. Mendes Bizerra Junior, D. Silva Silveira, M. Lencastre Pinheiro Menezes Cruz, and F. Araujo Wanderley, "A method for generation of tests instances of models from business rules expressed in ocl," *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 10, no. 5, pp. 2105–2111, 2012. [Online]. Available: http://dx.doi.org//10.1109/TLA.2012.6362355

[7] B. C. and M. A., "A framework for gui testing based on use case design," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 252–259. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2010.37

[8] K. Dyrkorn and F. Wathne, "Automated testing of non-functional requirements," in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ser. OOPSLA Companion '08. New York, NY, USA: ACM, 2008, pp. 719–720. [Online]. Available: http://doi.acm.org/10.1145/1449814.1449828

[9] S. R. Dalal and et al., "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 285–294. [Online]. Available: http://doi.acm.org/10.1145/302405.302640

[10] T. Straszak and M. Śmiałek, "Acceptance test generation based on detailed use case models," in *Advances in Software Development*, J. Swacha, Ed. PIPS, 2013, pp. 116–126.

[11] H. Kaindl, M. Śmiałek, P. Wagner, and et al., "Requirements specification language definition," ReDSeeDS Project, Project Deliverable D2.4.2, 2009, www.redseeds.eu.

[12] W. Nowakowski and et al., "Requirements-level language and tools for capturing software system essence," *Computer Science and Information Systems*, vol. 10, no. 4, pp. 1499–1524, 2013. [Online]. Available: http://dx.doi.org/10.2298/CSIS121210062N

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[14] I. M. Graham, "Task scripts, use cases and scenarios in object-oriented analysis," *Object-Oriented Systems*, vol. 3, no. 3, pp. 123–142, 1996.

[15] M. Śmiałek and et al., "Complementary use case scenario representations based on domain vocabularies," *Lecture Notes in Computer Science*, vol. 4735, pp. 544–558, 2007, mODELS'07. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75209-7_37

[16] "UML Testing Profile (UTP) Version 1.2," Object Management Group, Tech. Rep. formal/2013-04-03, Sep. 2012. [Online]. Available: http://www.omg.org/spec/UTP/1.2/

[17] M. Smialek and T. Straszak, "Facilitating transition from requirements to code with the ReDSeeDS tool," in *Requirements Engineering Conference (RE), 2012 20th IEEE International*. IEEE, 2012, pp. 321–322. [Online]. Available: http://dx.doi.org/10.1109/RE.2012.6345825

[18] M. Smialek and et al., "Translation of use case scenarios to Java code," *Computer Science*, vol. 13, no. 4, pp. 35–52, 2012. [Online]. Available: http://dx.doi.org/10.7494/csci.2012.13.4.35

[19] M. Smialek, W. Nowakowski, N. Jarzebowski, and A. Ambroziewicz, "From use cases and their relationships to code," in *MoDRE*. IEEE, 2012, pp. 9–18. [Online]. Available: http://dx.doi.org/10.1109/MoDRE.2012.6360084

[20] Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[21] D. North. (2006, Mar.) Introducing BDD. [Online]. Available: http://dannorth.net/introducing-bdd/