Computer Science and Information Systems 12(2):707–728 DOI: 10.2298/CSIS141217033S

Model-Driven Acceptance Test Automation Based on Use Cases

Tomasz Straszak, Michał Śmiałek

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

Abstract. Acceptance testing is highly dependent on the formulation of requirements, as the final system is tested against user requirements. It is thus highly desirable to be able to automate transition from requirements to acceptance tests. In this paper we present a model-driven approach to this issue, where detailed use case models are transformed into test cases. Importantly, our approach facilitates synchronising functional test cases with other types of tests (non-functional, domain logic, user interface) and introducing test data. This leads to a unified approach where requirements models of various kind drive the acceptance testing process. This process is parallel to the development process which also involves automatic transformations from requirements models to software development artefacts (models and code). To show validity of the approach we present a case study which uses a new tool called ReDSeT, that transforms requirements formulated in the Requirements Specification Language (RSL) into tests in a newly proposed Test Specification Language (TSL).
Keywords: acceptance testing, test generation, use cases, metamodel, model transformation.

1. Introduction and related work

Acceptance testing is the ultimate step in every software development effort [15]. Acceptance tests are part of the overall testing process [16], and are usually performed with high involvement of the customers who compare the final system's operations with the original contract between stakeholders and developers. This contract should be clear to both sides, so that the stakeholders can formulate their real needs and the developers can produce software compliant with these needs.

One of the most popular notations to specify functional requirements are use cases, introduced initially by Jacobson [12]. Use cases are represented through scenarios that define interactions between external actors and the system-to-be-built that lead to specific goals of certain value to the actors [5]. Use case scenarios can be specified using different notations, of which model-based ones have significant value [21]. Precise notations consisting of models with runtime semantics can be processed automatically to produce other software artifacts, including code [22].

Any system developed on the basis of use cases should normally be tested against these use cases. There exist several approaches to automate the process of obtaining test cases from use case models. The most basic ideas involve using activity notation for representing use case scenarios and generating test scripts expressed with some other modelling notation, like interaction models (see work by Gutiérrez et al. [11] and by Turner et al. [32]). Other proposed mechanisms involve analysis of use case contracts (pre- and post-conditions) to create more complex testing scenarios that consider changes of the system state (see work by Nebut et al. [17]). However, this approach works on a more formal level and uses sequence diagrams to denote scenarios. To understand such diagrams, a significant level of technical knowledge of UML is needed, which makes it arguably hard to inspect by the end-users.

An interesting approach was proposed by El-Attar and Miller [9]. It involves generating acceptance test scripts from textual use case scenarios combined with domain and robustness (class) models. The generated scripts contain test scenarios, description of the input and the expected results. In another, similar approach, Somé and Cheng create test cases through generating state machines [28]. State machines are also used by Jiang and Ding [13]. Their approach goes a step further by proposing to express use case scenarios using a formalised grammar with uniform subject-verb-object sentences.

Test cases based on use cases concentrate on verifying the functional requirements. Still, also other important aspects need to be assured: meeting the business rules, UI look-and-feel and various quality issues (performance, reliability etc.). These kinds of requirements should be verified by executing corresponding tests. A quite broad domain of test generation approaches is GUI-based, as surveyed by Banerjee et al. [1]. In the context of our work, an interesting approach was presented by Bertolini and Mota [3] who link GUI-based testing with use cases. Of other relevant approaches, we can mention work by Bizerra et al. [4] who introduce a method for generating tests from formally specified business rules associated with domain models and by Dyrkom and Wathne [8] who propose a method to automate test generation from non-functional requirements.

Most of the above mechanisms use model transformations forming the area of Model-Based Testing (MBT), which is an evolving technique for generating suites of test cases from software models of various kind [6, 20]. It should be noted that although different types of tests are generated from requirement models describing the same software system, usually they are not related. To verify different aspects of the system we need to use different methods which do not have direct interrelations. This paper aims at improving this situation. It focuses on automatic generation of different but interrelated types of tests. These tests are integrated through functional test cases obtained from model-based use case scenarios using an automated tool. The resulting test specification is expressed with a metamodel-based language called the Test Specification Language (TSL). In this language, every functional test case can integrate various other types of tests.

Such integrated test sets are generated on the basis of tightly interrelated requirements models that describe many aspects of the developed software system, which makes this idea MBT-compliant. The main idea of test automation is based on creating requirements specifications with a modelling language called the Requirements Specification Language (RSL) [14, 25]. Models in this language are used as input to the process illustrated in Figure 1 (see step 1).

The figure depicts two parallel model transformation paths that lead to acceptance testing of a delivered software application:

– software application production – application logic code is generated automatically on the basis of structured use case scenarios and interlinked domain models (step

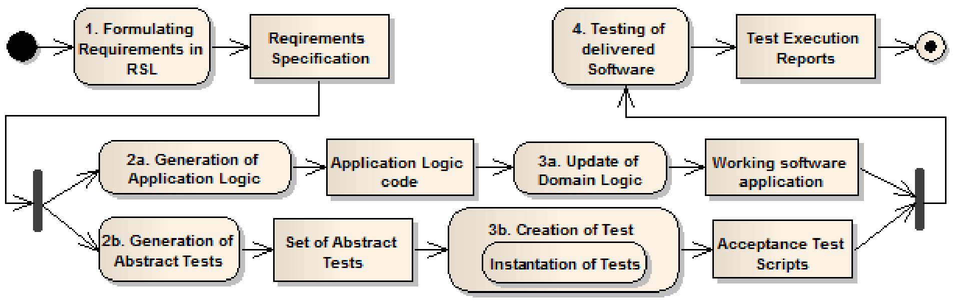

1. Formulating Requirements in RSL
Reqirements Specification
4. Testing of delivered Software
Test Execution Reports
2a. Generation of Application Logic
Application Logic code
3a. Update of Domain Logic
Working software application
2b. Generation of Abstract Tests
Set of Abstract Tests
3b. Creation of Test
Instantation of Tests
Acceptance Test Scripts

Fig. 1. Model-based software generation and test case generation as parallel processes

2a); having the application logic, domain logic code is updated manually according to given domain rules (step 3a);
– acceptance test script production – abstract tests are automatically generated on the basis of use cases and other related requirements (step 2b); test scenarios are then manually composed of instances of abstract tests (step 3b).

This process can be performed repeatedly according to incremental delivery of requirements. RSL provides means to denote use case scenarios very precisely thus significantly facilitating the generative steps of the two parallel processes. Additional information contained in scenario sentences (notions from the domain vocabulary) and other related requirements allow for generating other types of tests. All the tests generated on the basis of RSL-based requirements form a complete and coherent test suite for acceptance testing. The generation steps are performed using a tool called ReDSeT (Requirements-Driven Software Testing) written as a plug for the RSL tooling environment called ReDSeeDS (Requirements-Driven Software Development System) [24, 27].

This paper is a significant extension of our previous work [30], presented at the 5th International Workshop on Automating Test Case Design (FedCSIS). It introduces the TSL metamodel and related model transformation rules that allow for generating TSL models from RSL models. Moreover, it presents the process of obtaining the final test scripts from the generated abstract test cases and discusses a cases study performed to validate the approach.

2. 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, our solution is based on creating requirements models using RSL. The main feature of this language is the clear separation of descriptions of the system's visible behaviour from 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

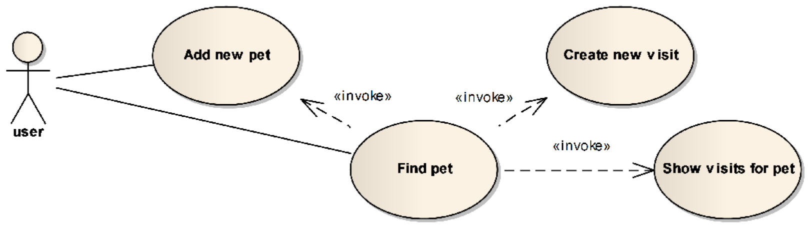
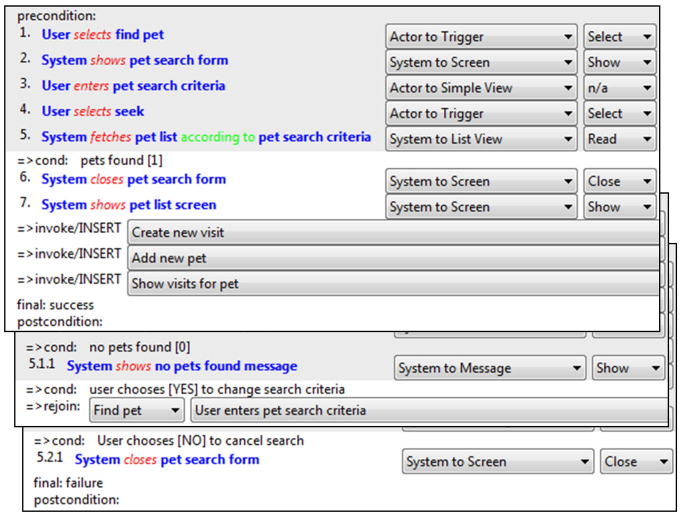


Fig. 2. Example use case model for the ‘Find pet’ use case



precondition:
1. User selects find pet — Actor to Trigger — Select
2. System shows pet search form — System to Screen — Show
3. User enters pet search criteria — Actor to Simple View — n/a
4. User selects seek — Actor to Trigger — Select
5. System fetches pet list according to pet search criteria — System to List View — Read

=>cond: pets found [1]
6. System closes pet search form — System to Screen — Close
7. System shows pet list screen — System to Screen — Show

=>invoke/INSERT Create new visit
=>invoke/INSERT Add new pet
=>invoke/INSERT Show visits for pet
final: success
postcondition:

=>cond: no pets found [0]
5.1.1 System shows no pets found message — System to Message — Show
=>cond: user chooses [YES] to change search criteria
=>rejoin: Find pet — User enters pet search criteria

=>cond: User chooses [NO] to cancel search
5.2.1 System closes pet search form — System to Screen — Close
final: failure
postcondition:

Fig. 3. Example scenarios for the ‘Find pet’ use case

for ordinary people who do not have technical expertise. The language has a precise specification of its syntax [14] and semantics [22] with methods of its use explained e.g. by Nowakowski et al. [19].

To illustrate RSL we present a fragment of a requirements model for the Pet Clinic system, adapted from the book by Śmiałek and Nowakowski [25]. Figures 2, 3 and 4 contain an elementary use case model with four use cases, scenarios for one of the use cases and an associated domain model. As we can notice in Figure 3, scenarios basically consist of numbered sentences in a simple ‘SVO(O)’ grammar, where sentences are sequences containing a Subject, a Verb and one or two Objects. These sentences are constructed using links to notions stored in the domain vocabulary presented in Figure 4.

Domain notions are referred to in scenario sentences through hyperlinks (see e.g.: *pet list*, *pet search form*, *find pet*) and are presented in a notion diagram that is similar to a class diagram. Relationships between notions, and the notion operations are defined automatically according to the scenario sentences in which these notions appear. Alternatively, they can be defined manually by the requirements engineer. Notions and their operations used in use case scenarios describe business domain logic and user interface elements.

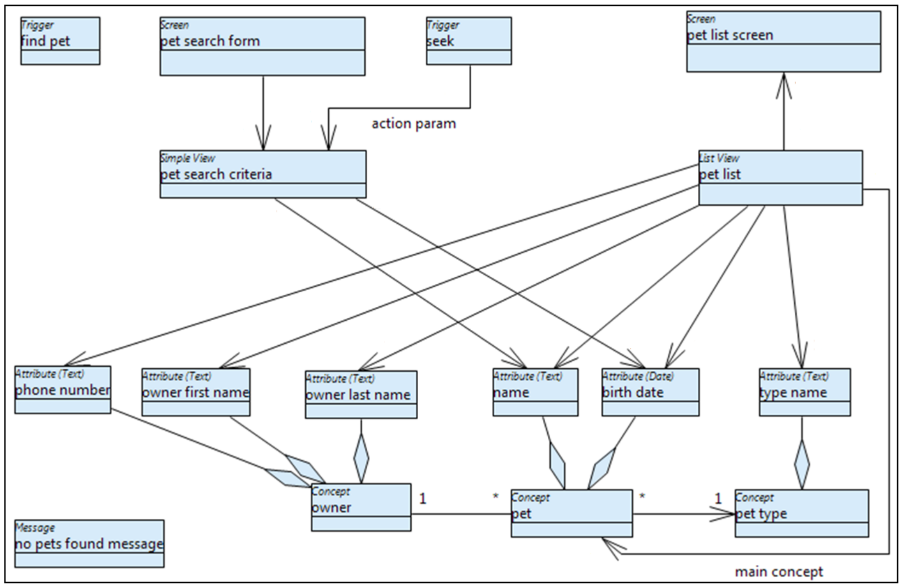


Fig. 4. Example domain model for the 'Find pet' use case

All the requirements in RSL can have relationships. To depict relations between use cases, a special invoke relationship is used (see Figure 2). This relationship has precise execution semantics. Each such relationship has to be associated with specific sentences in scenarios of the invoking use case that refer to the invoked use case. We can notice such sentences in the scenarios in Figure 3. These sentences have 'call' semantics and they mean that the respective invoked use case starts execution at this point. More details about semantics of invocations can be found in works by Nowakowski et al. [19] and Śmiałek et al. [26].

RSL offers much more capabilities than in the presented example. For instance, it allows for constructing tree-like package structures and introducing simple free-text requirements to describe business rules or quality (non-functional) aspects. Its formal specification [14] is based on a metamodel consisting of over 200 metaclasses. Detailed discussion of the metamodel and on developing transformations from RSL to UML and code can be found in the already mentioned book by Śmiałek and Nowakowski [25]. Instead, the following sections contains an overview of the testing language whose metamodel is derived from that of RSL. Next, we will present a transformation from RSL to this new testing language.

3. Test Specification Language

To define the proposed acceptance test suite and to ensure accurate and automatic transition from RSL-based requirements to tests, a new language, called the Test Specification Language (TSL) was developed. This language is based on a metamodel defined using EMF (Eclipse Modeling Framework) [29].

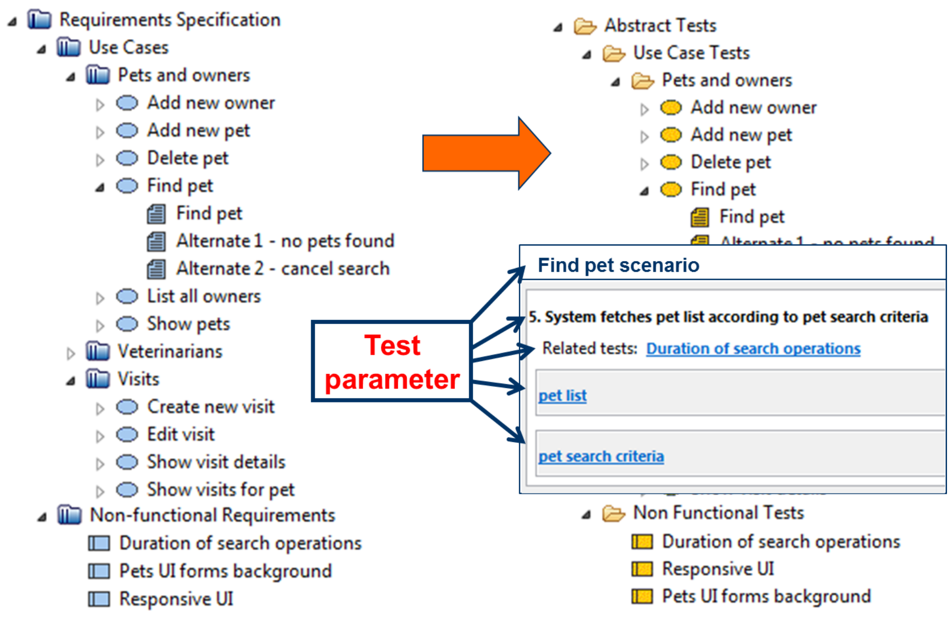


Fig. 5. Test specification derived from a requirements specification

The main idea of TSL is to provide notation for reusable tests, that are understandable for non technical people (end-users, clients, stakeholders, etc.) and precise enough to allow for verifying the software system in detail. All tests are grouped in a tree structure, named the Test Specification (see Figure 5 - right). This structure is an implementation of the metamodel shown in Figure 6. Each ***Test*** contained in the ***TestSpecification*** through a ***TestPackage***, represents a procedure for verification of a single software requirement. Such a verification is performed by processing through all the check points defined inside a test. Types of check points should be dedicated to each type of test as specialisations of ***TestParameter***s. The general attribute of this metaclass (*testingType*) determines whether the specific check point will be executed automatically or manually. Each specialisation of ***TestParameter*** should also include an attribute determining the check point result value, adequate for the test type.

The basic structure of a TSL test specification consists of two packages: Abstract Tests and Concrete Tests. The Abstract Tests package includes subtypes of ***AbstractTest***. This kind of tests are generated directly from the requirements specification and comprise mostly use case tests and notion-based tests, but can also contain tests of other types. They posses the *requirementUid* attribute that points to the source element in the requirements model. A use case test corresponds to a use case, and includes use case test scenarios generated from RSL use case scenarios, like the ones illustrated in Figure 5. The metamodel that represents this structure is shown in Figure 7.

A ***UseCaseTestScenario*** includes an initial condition (a precondition sentence) that must be met before the execution of actions described in this scenario and a final condition (a postcondition sentence) that describes the desired state of the system after the scenario is executed. Both the precondition and the postcondition are represented by the ***Condition***

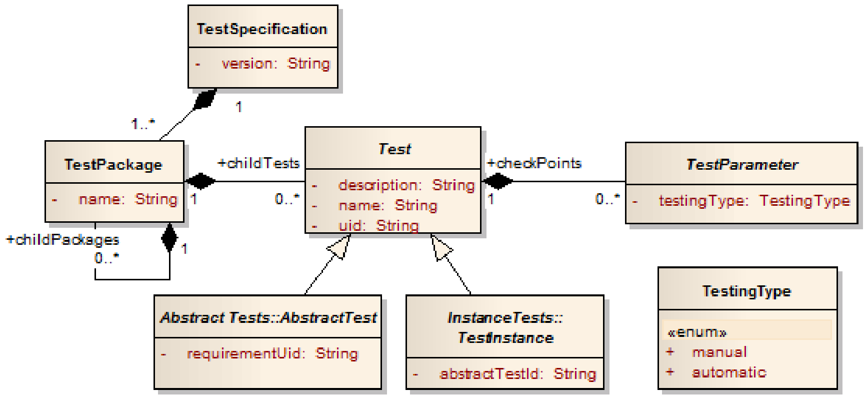


Fig. 6. TSL metamodel: test specification structure

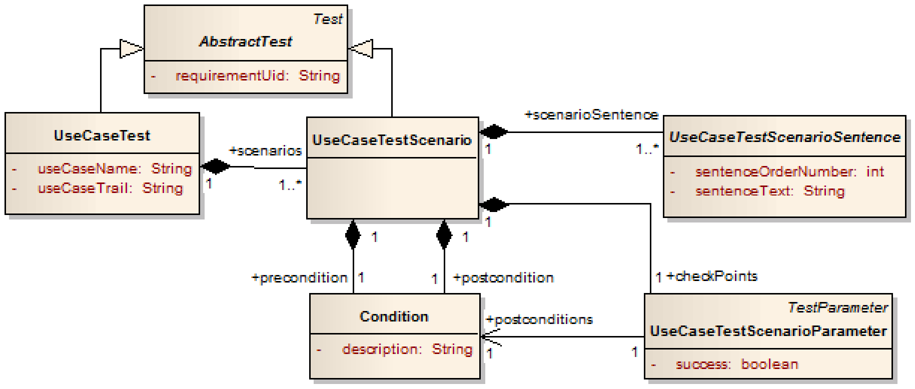


Fig. 7. TSL metamodel: use case tests

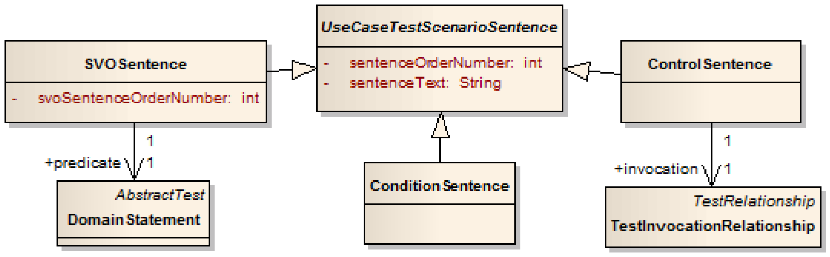


Fig. 8. TSL metamodel: use case test scenario sentences

metaclass. The check points for ***UseCaseTestScenario***s are postconditions pointed-at by ***UseCaseTestScenarioParameter***s.

Every ***UseCaseTestScenario*** consists of several ***UseCaseTestScenarioSentence***s. A dialogue between the primary actor and the system is a sequence of actions represented by sentences in a simple subject-verb-object (SVO) grammar (see Graham [10] for an

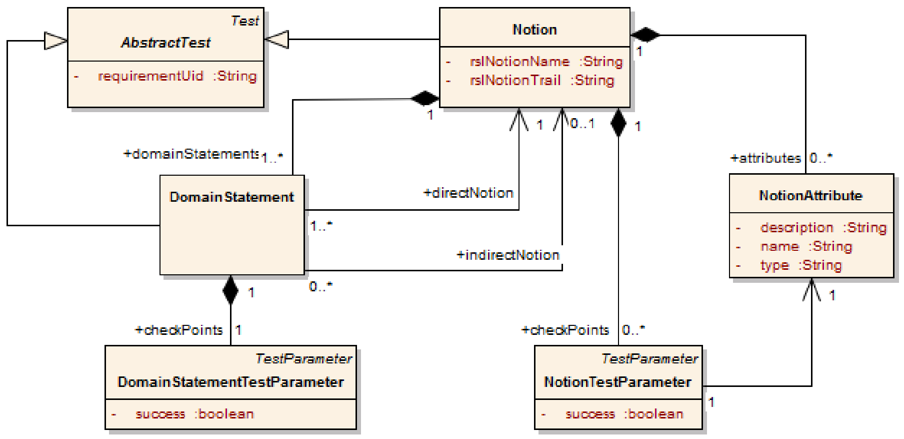


Fig. 9. TSL metamodel: notion tests

original idea). As shown in Figure 8, sentences describing single actions are represented by ***SVOSentence***s. This kind of sentence is described with a ***DomainStatement***, which forms its *predicate* (the verb-object part of an SVO sentence).

The ***DomainStatement*** metaclass, as shown in Figure 9, is a specialisation of ***AbstractTest*** and has check points represented by ***DomainStatementTestParameter***s. Each ***DomainStatement*** is owned by some ***Notion*** that is generated on the basis of an RSL notion. ***Notion*** tests can be verified according to ***NotionTestParameter***s that verify ***NotionAttribute***s.

In addition to action sentences, two additional sentence types were introduced: ***ConditionSentence***s and ***ControlSentence***s. Analogously to RSL, they are used in scenarios to express the flow of control between alternative scenarios of the same use case, as well as between scenarios of different use cases (see Śmiałek et al. [21]).

In addition to use case tests that verify behaviour of the system, tests of other types can verify the domain logic, the user interface, the non-functional aspects (performance, usability, etc,) or any other aspect of the system that is described through requirements. Figure 10 shows a taxonomy of such constructs which includes ***QualityTest***, ***GUITest***, ***DomainLogicTest*** and respective specialisations of the ***TestParameter*** metaclass. This metamodel template can be used for introducing other types of tests.

An important feature of a requirement specified with RSL, is the possibility to create relationships with other requirements. Due to generation of test specifications on the basis of these requirements, relationships between tests should be also created. Figure 11 presents metaclasses representing these relationships. The invocation relationships between use cases are translated to become relations between use case tests. This provides information on the steps of a use case test scenario and on conditions under which scenarios of appropriate other use case tests should be called. Relationships to requirements of other types are translated to relationships from use case tests to tests of other types.

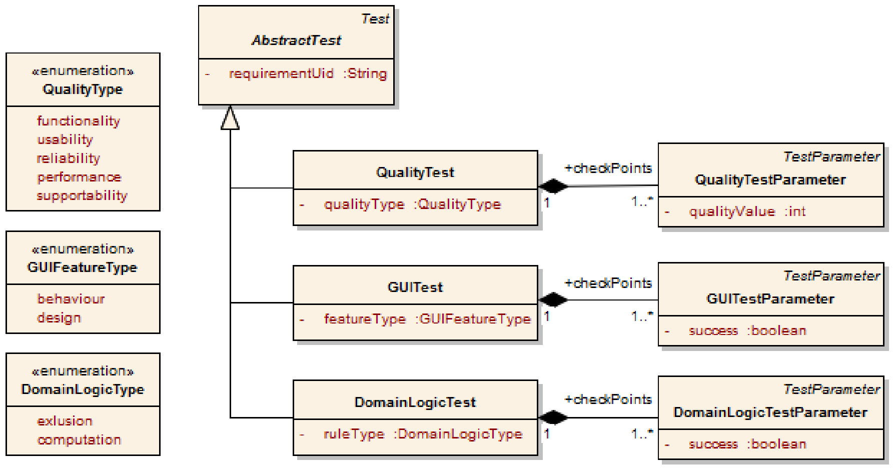


Fig. 10. TSL metamodel: tests of other types

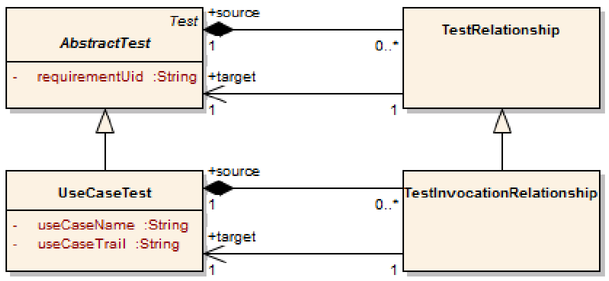


Fig. 11. TSL metamodel: test relationships

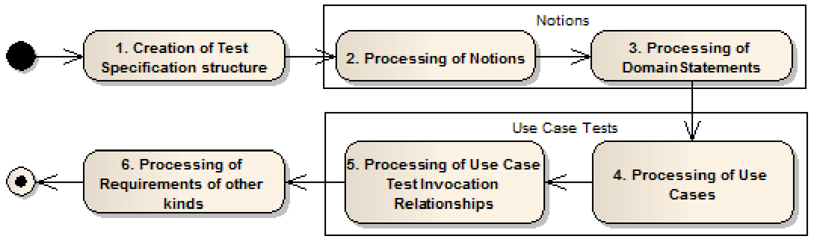

1. Creation of Test Specification structure
Notions
2. Processing of Notions
3. Processing of Domain Statements
Use Case Tests
4. Processing of Use Cases
5. Processing of Use Case Test Invocation Relationships
6. Processing of Requirements of other kinds

Fig. 12. RSL to TSL transformation procedure

4. Automatic test generation

Having RSL and TSL defined using metamodels, we can construct an automatic model transformation from requirements to tests. Moreover, further acceptance test composition is highly facilitated.

The transformation is performed in several steps. The top-level transformation steps of the transformation procedure are presented in Figure 12. At the beginning of the transformation, a new ***TestSpecification*** structure is created (step 1). The basic ***TestSpecification*** structure consists of a root node named using the scheme of "Software Case Name - date" and two child ***TestPackage***s named "Abstract Tests" and "Concrete Tests". The Abstract Tests package is constructed to reflect the source use case model structure. The Concrete Tests package is empty – Test Scenarios will be created here.

All ***SVOSentence***s within ***UseCaseTestScenario***s are composed of ***DomainStatement***s (e.g. 'enter pet search criteria') which are parts of specific ***Notion***s (e.g. 'pet search criteria'). For this reason, RSL notions should be processed first (step 2). For each RSL notion, a TSL ***Notion*** is created and placed in the proper package. The name, description and attached notion attributes are transferred, with the ***NotionTestParameter***s pointing at the ***NotionAttribute***s.

For all the ***Notion***s, domain statements taken from respective RSL notions are created (step 3). For each created ***DomainStatement*** a ***DomainStatementParameter*** is also created. The phrases contained in the RSL domain statement notions, used as the direct and the indirect object are found and pointed-at by the *directNotion* and the *indirectNotion* attributes.

Having processed the ***Notion***s and the ***DomainStatement***s, the transformation can now process use cases(step 4). For each RSL use case, a ***UseCaseTest*** is created and placed in a proper ***TestPackage*** within the use case test structure. The name and the description are transferred accordingly. All the scenarios contained in a RSL use case are transferred into a ***UseCaseTestScenario***. On the basis of the use case scenario's pre- and postcondition, adequate pre- and postconditions are created as ***Condition*** instances and attached to the respective ***UseCaseTestScenario***s. In addition, ***UseCaseTestScenarioParameter***s pointing at the postconditions are attached. Sentences of the processed RSL scenarios are transferred into appropriate instances of classes that specialise from ***UseCaseTestScenarioSentence***. For every ***SVOSentence***, the respective ***DomainStatement*** is found and a relation to the corresponding ***DomainStatement*** is created. For every control sentence, a ***TestInvocationRelationship*** is created with an empty ***UseCaseTest*** as its target.

Target ***UseCaseTest***s of the ***TestInvocationRelationship***s are set after all the use cases are transformed into ***UseCaseTest***s (step 5). For each ***TestInvocationRelationship*** contained in a ***ControlSentence***, a correct ***UseCaseTest*** is found and set.

At the end of the transformation (step 6), ***AbstractTest***s of other types are created. Each RSL requirement that is not a use case and is classified as a requirement of specific type (e.g. domain logic requirement, user interface requirement, non-functional requirement) is the basis for generating an ***AbstractTest***. This test supplements a ***UseCaseTest***, a ***UseCaseTestSenario***, a ***UseCaseTestSenarioSentence*** or a ***Notion***. RSL's relationships between use cases, notions and requirements of specific types are transformed into ***TestRelationship***s. As RSL currently does not support all the requirements types, only non-functional requirements are automatically transformed into ***QualityTest***s.

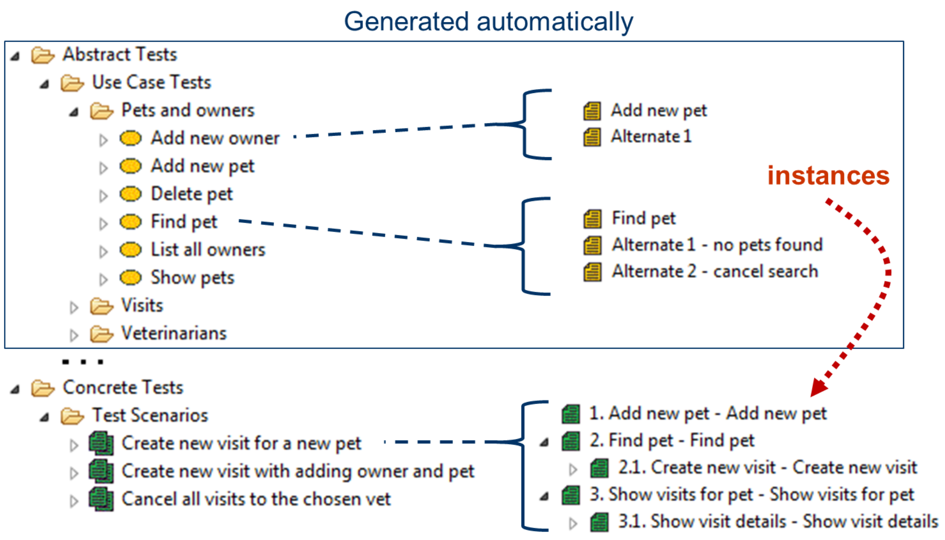

Generated automatically
Abstract Tests
Use Case Tests
Pets and owners
Add new owner
Add new pet
Delete pet
Find pet
List all owners
Show pets
Visits
Veterinarians
Add new pet
Alternate 1
Find pet
Alternate 1 - no pets found
Alternate 2 - cancel search
instances
...
Concrete Tests
Test Scenarios
Create new visit for a new pet
Create new visit with adding owner and pet
Cancel all visits to the chosen vet
1. Add new pet - Add new pet
2. Find pet - Find pet
2.1. Create new visit - Create new visit
3. Show visits for pet - Show visits for pet
3.1. Show visit details - Show visit details

Fig. 13. Test scenarios composed of test cases

5. Test instantiation

A scenario of a use case test determines conditions, steps and check points that will verify implementation for a given use case. These elements will be used to perform acceptance tests after placing them in test scenarios and assigning specific test data values. Test scenarios are grouped into second-level packages within the general structure of a ***TestSpecification*** (see Figure 13). They are defined by a test engineer as a set of ordered instances of use case test scenarios, that are called "test cases". A test case describes a single procedure for verifying system's functionality. It is composed of ordered sentences describing actions, conditions and invocations of other test cases. This structure is represented by the metamodel shown in Figure 14.

To examine the result of executing a whole ***TestScenario***, a ***TestScenarioCheckPoint*** is used. The *expectedResult* attribute is used to denote the success or failure of scenario execution according to some given conditions. The overall result depends on the results of individual ***TestCase***s, forming the steps of the ***TestScenario***. Each ***TestCase*** is examined in relation to the ***TestCaseCheckPoint*** that refers to the *postcondition* of the given ***TestCase***.

TestCases are composed of ***TestCaseSentence***s, as illustrated in Figure 15. Actions taken by the main actor or by the system are represented by ***SVOSentenceInstance***s and contain ***SVOSentenceInstanceCheckPoint***s. Such check points represent places where execution of related actions needs to be examined. The result of checking an ***SVOSentenceInstance*** depends on the states of ***DomainObject***s that are determined by respective ***DomainObjectCheckPoint***s. These check points focus on the values of the ***DomainObjectAttribute***s.

The metaclasses ***TestScenario***, ***TestCase***, ***SVOSentenceInstance***, and ***DomainObject*** are specialisations of ***TestInstance***. Thus, they can have ***TestInstances*** of other types at-

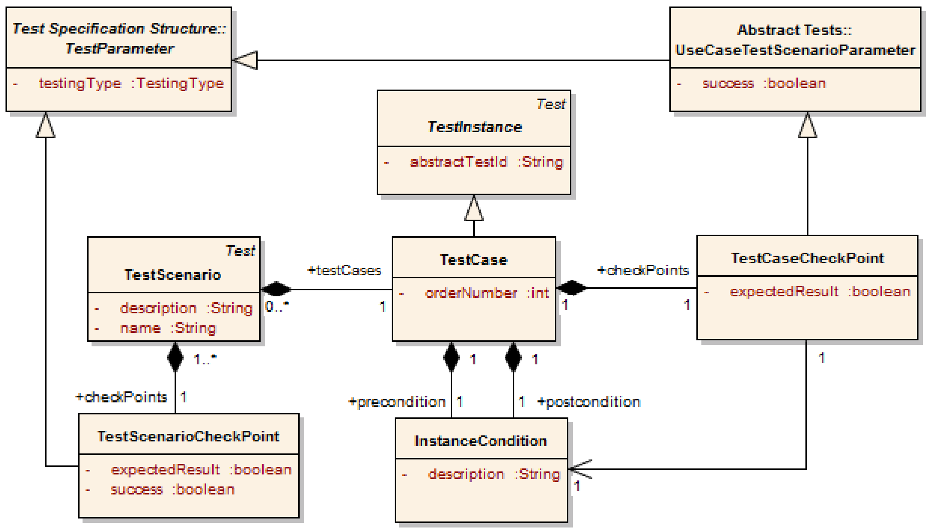


Fig. 14. TSL metamodel: test scenario and test cases

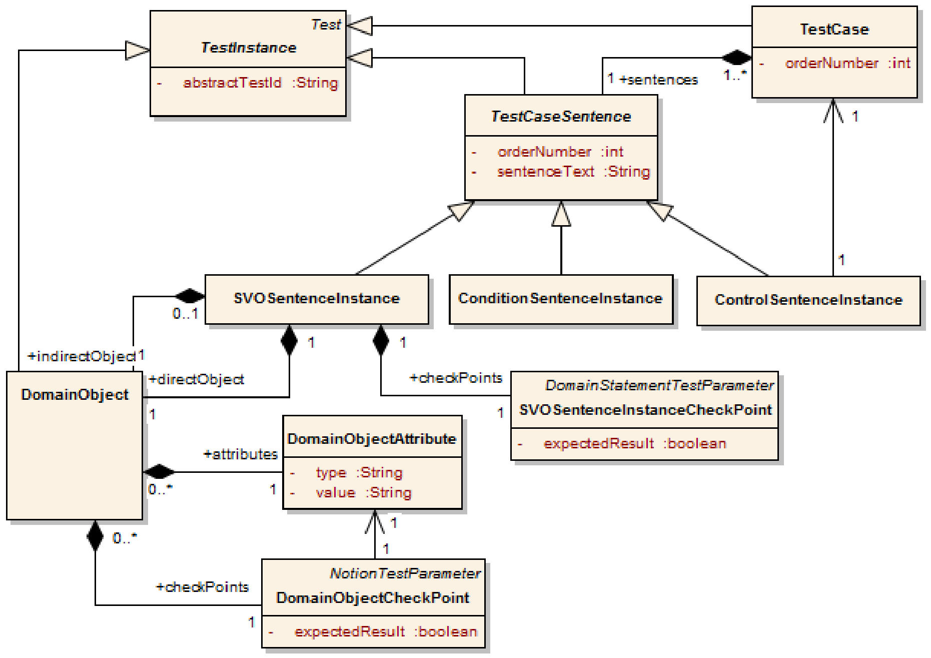


Fig. 15. TSL metamodel: test case sentences and domain objects

tached. By analogy with the abstract tests metamodel, we can declare ***NonFunctional-***

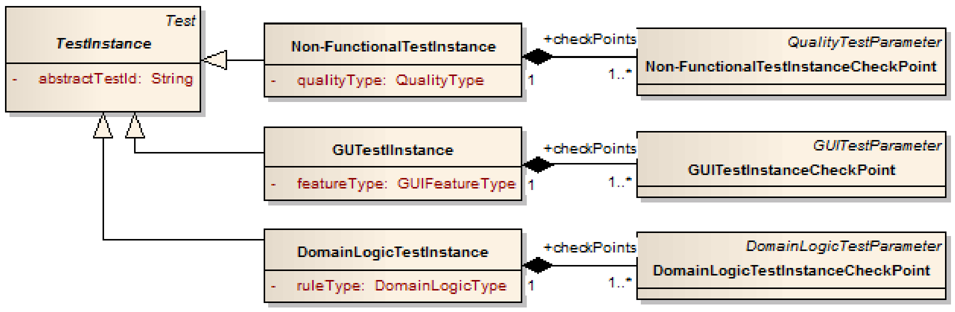

Test
TestInstance
- abstractTestId: String
Non-FunctionalTestInstance
- qualityType: QualityType
+checkPoints
1
1..*
QualityTestParameter
Non-FunctionalTestInstanceCheckPoint
GUITestInstance
- featureType: GUIFeatureType
GUITestParameter
GUITestInstanceCheckPoint
DomainLogicTestInstance
- ruleType: DomainLogicType
DomainLogicTestParameter
DomainLogicTestInstanceCheckPoint

Fig. 16. TSL metamodel: test instances of other types

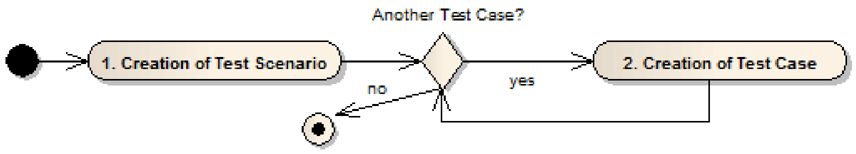


Fig. 17. Test Scenario creation procedure

TestInstances, ***GUITestInstance***s and ***DomainLogicTestInstance***s, as shown in Figure 16. ***TestInstance***s of these other types are automatically created during execution of the instantiation procedure. They have their own check points, and their results affect the results of their parent tests.

Test scenarios constructed with test cases also build the context for the test data. The initial test data values are set by the test engineer as the precondition values of the test scenarios. Test data values describe basic business objects as well as GUI elements. Test data in scope of one test scenario is passed between test cases as its pre- and postcondition values. The test data values change according to the functionality and the business logic that is being tested. It can be noted that although 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 under test.

The instantiation procedure of a use case test scenario consists of adding consecutive test cases. This simple process is illustrated in Figure 17. First, we create a ***TestScenario***, give it a *name* and a *description*, and situate it in an instance of a ***TestPackage***. Then, we attach to it consecutive ***TestCase***s. The available test cases depend on compatibility of their pre- and postconditions.

For each newly added ***TestCase***, an automatic instantiation procedure is performed. The procedure is presented in Figure 18. At its beginning the ***TestCase*** is assigned a consecutive number. For a nested ***TestCase***, its number is segmented (e.g.: *2.3.1*). The *name* of the chosen ***UseCaseTestScenario*** and the *description* of the corresponding ***UseCaseTest*** are transferred into the ***TestCase*** (step 1). The same is done for the ***UseCaseTestScenario***'s *precondition* and *postcondition* (step 2).

Having a given ***TestCase*** created, appropriate ***SVOSentenceInstance***s, ***ConditionSentenceInstance***s and ***InvokeSentenceInstance***s are created. All the ***TestCaseSentence***s (step 3) are transferred with their *orderNumber*s and *sentenceText*s. All ***AbstractTest***s

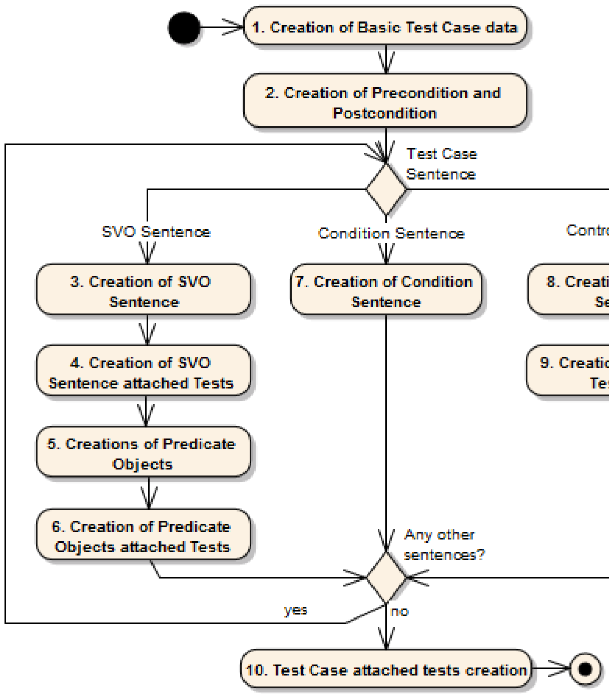

1. Creation of Basic Test Case data
2. Creation of Precondition and Postcondition
Test Case Sentence
SVO Sentence
Condition Sentence
Control Sentence
3. Creation of SVO Sentence
7. Creation of Condition Sentence
8. Creation of Control Sentence
4. Creation of SVO Sentence attached Tests
9. Creation of invoked Test Case
5. Creations of Predicate Objects
6. Creation of Predicate Objects attached Tests
Any other sentences?
yes
no
10. Test Case attached tests creation

Fig. 18. Test Case instantiation procedure

of other types related to the ***DomainStatement*** pointed-at by the *predicate* relation are transferred into adequate ***TestInstance*** specialisations. These ***TestInstance***s are inserted as parts of appropriate ***SVOSentenceInstance***s (step 4). In the next step, *directObject*s and *indirectObject*s of sentence predicates are created as ***DomainObject***s on the basis of direct and indirect ***Notion***s of the appropriate ***DomainStatement*** (step 5). All ***AbstractTest***s related to ***Notion***s pointed-at by the ***Domain statement***s' *directNotion* and *indirectNotion* relations are transferred into adequate ***TestInstance*** specialisation instances. These ***TestInstance***s of other types are contained as *directObject*s or *indirectObject*s (step 6).

In case of condition sentences, respective ***ConditionSentenceInstance***s are generated with their *orderNumber* and *sentenceText* preserved (step 7). Also, appropriate ***ControlSentenceInstance***s are generated with appropriate *orderNumber* and *sentenceText* (step 8). Depending on the test engineer's decision, a nested ***TestCase*** can be created on the basis of the ***UseCaseTest*** related through a ***TestInvocationRelationship*** to the currently processed ***UseCaseTest*** (step 9). If a use case test invocation is used, one of the invoked ***UseCaseTestScenario***s has to be instantiated recursively.

After creating test instances related to individual sentences, the procedure creates ***TestInstance***s for all the ***AbstractTest***s related to the processed ***UseCaseTestScenario*** (step 10). These ***TestInstance***s are contained in the appropriate ***TestCase***s. For each ***TestInstance***, specialisation check points are created as subtypes of relevant test parameters.

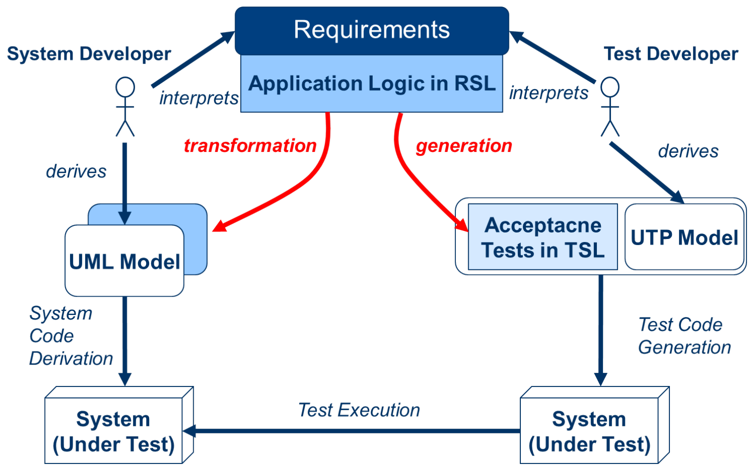


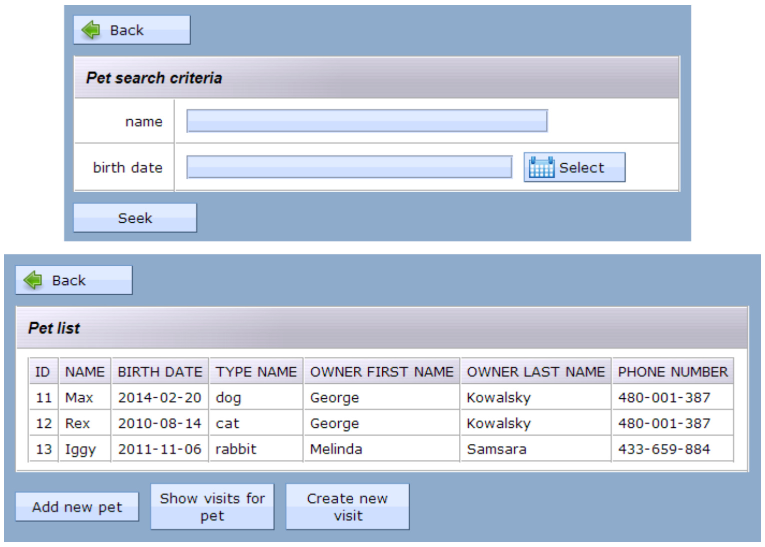
Fig. 19. TSL as a complement for the UML Testing Profile

6. 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. Specifically, it can be used in conjunction with UML-based testing. UML is commonly used for software modelling but it lacks constructs specific to software testing. To supplement UML in the area of testing, the UML Testing Profile (UTP) was introduced by the OMG (Object Management Group) [33]. It uses typical UML profiling mechanisms and offers a domain-independent specification of various test concepts based on the UML notation. UTP allows for creation, documentation, visualization, specification and exchange of model-based test specifications.

The UTP-based validation can be used in conjunction with TSL-based testing. Figure 19 presents a relevant scenario. Starting from requirements, a system developer delivers UML models, which are the basis for developing a given 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 [34]. 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 created. These traces facilitate linking of UTP models with corresponding acceptance tests in TSL through appropriate UML models. This allows to deliver a complete test suite, where overall acceptance tests of a complex software system are expressed in TSL and tests of other software project artefacts are expressed in UTP.



ID	NAME	BIRTH DATE	TYPE NAME	OWNER FIRST NAME	OWNER LAST NAME	PHONE NUMBER
11	Max	2014-02-20	dog	George	Kowalsky	480-001-387
12	Rex	2010-08-14	cat	George	Kowalsky	480-001-387
13	Iggy	2011-11-06	rabbit	Melinda	Samsara	433-659-884

Fig. 20. Example screens produced from requirements in RSL

7. Case study

To validate the presented approach, we have conducted a case study. It is based on extended Pet Clinic requirements briefly introduced in Figures 2, 3 and 4. Figure 20 shows example screens produced from the domain model in Figure 4. The upper screen reflects the ‘pet search form’ element with the associated ‘pet search criteria’ and attributes. Analogously, the lower screen reflects the ‘pet list’ element. These two screens should be presented to the user at appropriate times during use case execution (see sentences 2 and 5 in Figure 3).

To perform the case study, we have developed a tool called ReDSeT (Requirements Driven Software Testing). It is based on the Eclipse Rich Client Platform. It is closely integrated with the ReDSeeDS tool (www.redseeds.eu [27]) which provides advanced editors for RSL models (use cases, scenarios, notions and other requirements). The generated test specifications can be included in the same Eclipse project as the source requirements specification and code. This enables integration of activities at different stages of the software development project.

The ReDSeT tool is designed as a set of integrated plug-ins, which are responsible for: automatic generation of tests, test management, use case test viewing, test scenario editing, composing sequences of test cases and test data value editing. Test viewers and test data value editors for the tests of other types can be additionally attached to the tool.

For the RSL to TSL model transformation, a dedicated transformation engine was developed. It reads the RSL model stored in the ReDSeeDS repository (it uses a Java-based graph technology [31]), executes the RSL to TSL transformation rules implemented in Java and writes the TSL model into an EMF-based repository. Due to the use of the EMF technology [29], 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. For the test specification repository, XML files are used. This provides for technical capabilities to easily extract test scripts when executing acceptance tests.

During the cases study, we first transformed the Pet Clinic requirements specification into a test specification using the ReDSeT tool. When the automatic transformation was complete, it was possible to manage the generated test specification organised in a tree structure using the Test Specification Browser and dedicated editors available through the ReDSeT perspective of the ReDSeeDS tool. Figure 5 shows parts of the structure of the generated abstract tests corresponding to the requirements model for the Pet Clinic system. The created use case tests and use case test scenarios could be reviewed in the Test Editor. A dedicated Detailed Test View can be used for viewing test check points.

Having abstract tests generated, we created test scenarios composed of instantiated use case test scenarios, using dedicated wizards. Figure 13 shows the 'Create new visit for a new pet' test scenario structure in the Test Specification Browser. The wizards allowed to make decisions on including optionally invoked use cases in the test scenario. In the example test scenario, test cases '2.1 Create new visit - Create new visit' and '3.1 Show visit details - Show visit details' have been chosen to be instantiated as being invoked from the respective two invoking use cases. Test cases could be viewed in the Test Editor, while test values contained in test cases and included test of all types were edited in the Detailed Test View.

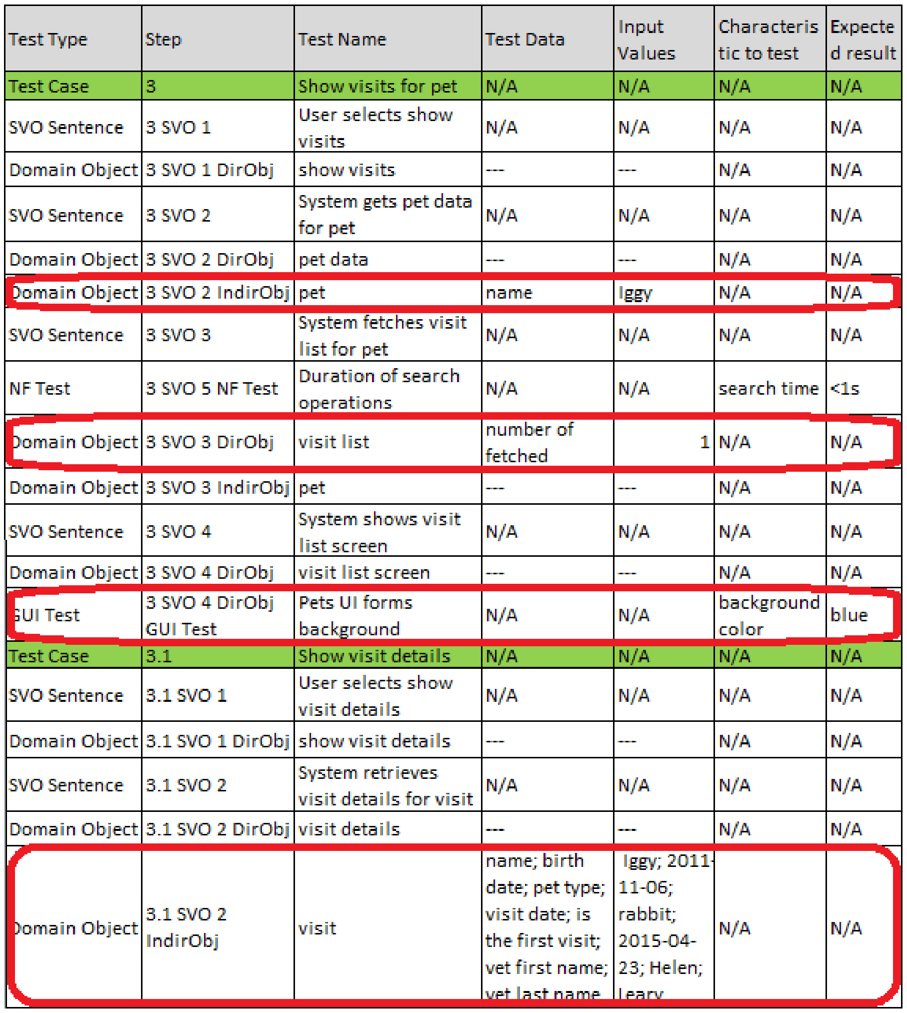
In order to perform acceptance tests according to test scenarios defined in the ReDSeT tool, test execution scripts were generated. These test scripts contain detailed steps for the testers in the form of structured text. Each line represents one test with its name, description, input data values and expected state of the system for the specified elements. Part of a test execution script for the 'Create new visit for a new pet' is presented in Figure 21 in the form of a CSV text file. The rows that represent steps in a test scenario have filled background and are shown as numbered test cases.

Each sentence in a test case is numbered with a test case number and an SVO sentence number. Additional tests are included below respective test case sentences as shown in Figure 21. For example, sentences *3 SVO 2* and *3.1 SVO 2* have additional tests which define test data values consistent with appropriate domain elements ('pet' and 'visit') attached to these particular sentences. A Non Functional Test is attached to the step 'Systems fetches visit list for pet' to examine performance of the search operation. In turn, a GUI test is presented below the step *3 SVO 4* and is attached to the step's direct object ('visit list screen') which is a UI element.

On the basis of such test execution scripts, we have verified the developed Pet Clinic software system. The results of each test step were noted in an additional column. In the cases when tests have failed, the corresponding requirement could be easily found by tracing to the appropriate requirements element. The implementation units were precisely located then by examining traces to code, as described in the work by Śmiałek et al. [23].

During the case study, several test scenarios, like: 'Create new visit for a new pet', 'Create new visit with adding an owner and a pet' or 'Cancel all visits to the chosen vet' were created manually and automatically with the ReDSeT tool from the Pet Clinic requirements specification.

The most visible advantage of using the ReDSeT tool in comparison to a fully manual approach is a visible reduction of effort used to prepare tests. We did not need to manually re-write (copy-paste with significant modifications) use case scenarios into test scenarios.



Test Type	Step	Test Name	Test Data	Input Values	Characteristic to test	Expected result
Test Case	3	Show visits for pet	N/A	N/A	N/A	N/A
SVO Sentence	3 SVO 1	User selects show visits	N/A	N/A	N/A	N/A
Domain Object	3 SVO 1 DirObj	show visits	---	---	N/A	N/A
SVO Sentence	3 SVO 2	System gets pet data for pet	N/A	N/A	N/A	N/A
Domain Object	3 SVO 2 DirObj	pet data	---	---	N/A	N/A
Domain Object	3 SVO 2 IndirObj	pet	name	Iggy	N/A	N/A
SVO Sentence	3 SVO 3	System fetches visit list for pet	N/A	N/A	N/A	N/A
NF Test	3 SVO 5 NF Test	Duration of search operations	N/A	N/A	search time	<1s
Domain Object	3 SVO 3 DirObj	visit list	number of fetched	1	N/A	N/A
Domain Object	3 SVO 3 IndirObj	pet	---	---	N/A	N/A
SVO Sentence	3 SVO 4	System shows visit list screen	N/A	N/A	N/A	N/A
Domain Object	3 SVO 4 DirObj	visit list screen	---	---	N/A	N/A
GUI Test	3 SVO 4 DirObj GUI Test	Pets UI forms background	N/A	N/A	background color	blue
Test Case	3.1	Show visit details	N/A	N/A	N/A	N/A
SVO Sentence	3.1 SVO 1	User selects show visit details	N/A	N/A	N/A	N/A
Domain Object	3.1 SVO 1 DirObj	show visit details	---	---	N/A	N/A
SVO Sentence	3.1 SVO 2	System retrieves visit details for visit	N/A	N/A	N/A	N/A
Domain Object	3.1 SVO 2 DirObj	visit details	---	---	N/A	N/A
Domain Object	3.1 SVO 2 IndirObj	visit	name; birth date; pet type; visit date; is the first visit; vet first name; vet last name	Iggy; 2011-11-06; rabbit; 2015-04-23; Helen; Leary	N/A	N/A

Fig. 21. Test execution script - attached test and SVO sentence object test

We just had to pick coarse-grained steps in the test scenario and the rest was done by the tool. Due to requirements relationships transferred into the test specification, there was no need to seek for all the related requirements to enclose them in the test scenarios. The ReDSeT tool automatically transformed requirements of all types into abstract tests and during instantiation they became part of test cases. Moreover, based on the tight coupling of the behavioural specification with the separate domain vocabulary, the ReDSeT tool could include all the required test data to be examined during testing. We could notice that the test specification created purely manually had problems with determining all the necessary paths that needed to be traversed to verify the developed system. By contrast, through using the ReDSeT tool we could ensure that all the interrelated use cases and other requirements are consistently traversed.

The only drawback we have noticed when using the ReDSeT tool was that complete tests could be executed only after completing a significant portion of the requirements specification. Lacking sentences in test scenarios could be completed only by extending the requirements model and re-transforming it into abstract tests and then recomposing test scenarios. Thus knowledge and good management of the source RSL model was crucial to successful test execution. Moreover, we have also noticed that the process could significantly benefit from implementing partial generation and linking algorithms similarly to those used for incremental code generation.

8. Conclusion

Our conclusion of the case study is that the proposed solution can be successfully used for quick creation of tests that focus on intensive interaction between software systems and their users. The obtained test scenarios define high quality instructions that can be easily followed to verify the developed system from both functional and non-functional points of view. Moreover, it assures that the created tests are coherent in terms of data that needs to be provided and expected at the various check-points in the test scenarios.

The proposed idea and the ReDSeT tool offer a complete solution for creating acceptance tests suitable for interactive systems. The basis for creating sets of test scenarios are detailed use case models with precisely specified scenarios. Requirements defined using RSL significantly facilitate automatic test generation, and TSL allows for expressing interrelated tests of different types in a way that should be comprehensible to the audience responsible for acceptance testing.

The automatically generated tests can be re-used in various test scenarios. Preparation of such generated tests can be done in parallel to formulation of requirements thus facilitating various test-driven approaches. Every change in requirements can be quickly represented in tests through their re-generation, retaining high level of detail of the changed requirements.

It can be noted that the proposed method consists in black box testing and is independent of the implementation technology of the system under test. Since RSL and TSL are defined using metamodels, various other models can be generated from RSL and TSL models. Also, traces from requirements to test cases can be used for automatic processing. For example, test coverage reports can be generated which also associates this approach with the problems of regression testing. Analysis of various possibilities and creation of appropriate methods and tools will be subject to further research.

Another area of further research is to use RSL, TSL and their transformations [26] as an implementation of Test Driven Development (TDD) [2] and Behaviour Driven Development (BDD) [18]. These ideas assume that software development is driven by tests developed already at early stages of the software increments. The proposed solution seems to have high potential to facilitate such approaches by making the whole process of test creation very agile through automation of the transition from the user needs (cf. models in RSL) to the tests that drive development.

Since TSL is built using an EMF-compliant metamodel and the ReDSeT tool is constructed as an Eclipse plug, the solution can be easily extended. To support other types of tests, the metamodel and appropriate editor plug-ins should be developed. In the future, it is planned to extend the solution by including detailed tests for the business logic and the graphical user interface. It is also planned to extend the tool with the mechanisms for generating test scripts in formats acceptable by test automation tools like e.g. IBM Rational Functional Tester [7] or Selenium. This would allow for even higher levels of automation in black-box testing.

References

1. Banerjee, I., Nguyen, B., Garousi, V., Memon, A.: Graphical user interface (GUI) testing: Systematic mapping and repository. Information and Software Technology 55(10), 1679–1694 (2013)
2. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
3. Bertolini, C., Mota, A.: A framework for GUI testing based on use case design. In: Proc. Third International Conference on Software Testing, Verification, and Validation: Workshops. pp. 252–259 (2010)
4. Bizerra Junior, E.M., Silva Silveira, D., Lencastre Pinheiro Menezes Cruz, M., Araujo Wanderley, F.: A method for generation of tests instances of models from business rules expressed in OCL. Latin America Transactions, IEEE (Revista IEEE America Latina) 10(5), 2105–2111 (2012)
5. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2000)
6. Dalal, S.R., et al.: Model-based testing in practice. In: Proc. 21st International Conference on Software Engineering (ICSE '99). pp. 285–294. ACM (1999)
7. Davis, C., Chirillo, D., Gouveia, D., et al.: Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource. IBM Press (2009)
8. Dyrkorn, K., Wathne, F.: Automated testing of non-functional requirements. In: Companion 23rd Conference on Object-Oriented Programming Systems Languages and Applications. pp. 719–720. ACM (2008)
9. El-Attar, M., Miller, J.: Developing comprehensive acceptance tests from use cases and robustness diagrams. Requirements Engineering 15(3), 285–306 (2010)
10. Graham, I.M.: Task scripts, use cases and scenarios in object-oriented analysis. Object-Oriented Systems 3(3), 123–142 (1996)
11. Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J.: An approach to generate test cases from use cases. In: Proc. 6th International Conference on Web Engineering. pp. 113–114. ACM (2006)
12. Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley (1992)
13. Jiang, M., Ding, Z.: Automation of test case generation from textual use cases. In: 4th International Conference on Interaction Sciences. pp. 102–107 (2011)
14. Kaindl, H., Śmiałek, M., Wagner, P., et al.: Requirements specification language definition. Project Deliverable D2.4.2, ReDSeeDS Project (2009), http://www.redseeds.eu/
15. Marciniak, J.J., Shumskas, A.: Encyclopedia of Software Engineering, chap. Acceptance Testing. Wiley, 2 edn. (2002)
16. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3 edn. (2011)
17. Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.M.: Automatic test generation: A use case driven approach. IEEE Transactions on Software Engineering 32, 140–155 (2006)
18. North, D.: Introducing BDD. Better Software Magazine (Mar 2006)
19. Nowakowski, W., Śmiałek, M., Ambroziewicz, A., Straszak, T.: Requirements-level language and tools for capturing software system essence. Computer Science and Information Systems 10(4), 1499–1524 (2013)

20. Shirole, M., Kumar, R.: UML behavioral model based test case generation: A survey. SIGSOFT Softw. Eng. Notes 38(4), 1–13 (2013)
21. Śmiałek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T.: Complementary use case scenario representations based on domain vocabularies. Lecture Notes in Computer Science 4735, 544–558 (2007), MODELS'07
22. Śmiałek, M., Jarzebowski, N., Nowakowski, W.: Runtime semantics of use case stories. In: 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 159–162 (2012)
23. Śmiałek, M., Jarzebowski, N., Nowakowski, W.: Translation of use case scenarios to Java code. Computer Science 13(4), 35–52 (2012)
24. Śmiałek, M., Kalnins, A., Ambroziewicz, A., Straszak, T., Wolter, K.: Comprehensive system for systematic case-driven software reuse. Lecture Notes in Computer Science 5901, 697–708 (2010), SOFSEM'10
25. Śmiałek, M., Nowakowski, W.: From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice. Springer (2015)
26. Śmiałek, M., Nowakowski, W., Jarzebowski, N., Ambroziewicz, A.: From use cases and their relationships to code. In: Second IEEE International Workshop on Model-Driven Requirements Engineering. pp. 9–18. IEEE (2012)
27. Śmiałek, M., Straszak, T.: Facilitating transition from requirements to code with the ReDSeeDS tool. In: 20th IEEE International Requirements Engineering Conference. pp. 321–322. IEEE (2012)
28. Somé, S.S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: Proc. 2008 ACM Symposium on Applied Computing. pp. 724–729. ACM (2008)
29. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley, 2 edn. (2009)
30. Straszak, T., Śmiałek, M.: Automating acceptance testing with tool support. In: 2014 Federated Conference on Computer Science and Information Systems. pp. 1569–1574 (2014)
31. Team, J.: JGraLab: The Java Graph Laboratory. http://jgralab.uni-koblenz.de
32. Turner, D.A., Park, M., Kim, J., Chae, J.: An automated test code generation method for web applications using activity oriented approach. In: Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 411–414 (2008)
33. UML Testing Profile (UTP) Version 1.2. Tech. Rep. formal/2013-04-03, Object Management Group (2012)
34. Zander, J., Dai, Z., Schieferdecker, I., Din, G.: From U2TP models to executable tests with TTCN-3: An approach to model driven testing. Lecture Notes in Computer Science 3502, 289–303 (2005), TestCon'05

Tomasz Straszak is a researcher interested in software modeling, requirements engineering and test engineering. He currently finalises his PhD work at the Warsaw University of Technology. He gained professional experience in telco and banking sectors working as a system/business analyst, software and solution architect and programmer.

Michał Śmiałek currently holds the position of a Professor. He obtained a habilitation (higher doctorate) degree in informatics from the Warsaw Military University and has graduated the Warsaw University of Technology (MSc and PhD) and the University of Sheffield (MSc). Prof. Smialek has more than 20 years of experience in software development mainly using object-oriented methods. For several years he worked in the industry as a software developer and project manager. He teaches software modelling and requirements engineering in academia and for the major Polish companies. He published two books: on UML modelling and model-driven requirements, and over 70 articles in national and international refereed journals and conference proceedings. He is a member of program committees of international conferences in the area of software engineering, and did reviews for major software engineering journals. His research interests include metamodelling, model transformations, scenario-based requirements engineering and object-oriented development methods.

Received: December 17, 2014; Accepted: June 19, 2015.