A Model-Driven approach for functional test case generation

J.J. Gutiérrez, M.J. Escalona, M. Mejías

Department of Computer Languages and Systems, ETS Ingeniería Informática, IWT2 Research Group, University of Seville, Av. Reina Mercedes S/N, Seville, Spain

Keywords:

Software quality assurance

Model-Driven testing

Early testing

A B S T R A C T

Test phase is one of the most critical phases in software engineering life cycle to assure the final system quality. In this context, functional system test cases verify that the system under test fulfills its functional specification. Thus, these test cases are frequently designed from the different scenarios and alternatives depicted in functional requirements. The objective of this paper is to introduce a systematic process based on the Model-Driven paradigm to automate the generation of functional test cases from functional requirements. For this aim, a set of metamodels and transformations and also a specific language domain to use them is presented. The paper finishes stating learned lessons from the trenches as well as relevant future work and conclusions that draw new research lines in the test cases generation context.

1. Introduction

Software quality assurance is one of the most critical steps in software development. Validation and verification techniques were proposed in the software research context to ensure quality and most of them are mainly organized into test phase. Test phase frequently allows us to group each activity oriented toward validating each aspect of our software results, from a small piece of code (with unit tests) to a total piece of the final system (with acceptance user tests)(Binder, 2000).

However, despite its relevance, test phase is frequently planned with very few resources, without an expert group of tests and a reduced group of techniques or tools that help support it (Shah et al., 2014; Huda et al., 2015). Besides, test phase is frequently a good candidate to keep up a delay, with resources or time reductions, when a software project is delayed (Li et al., 2010; Felderer and Ramler, 2014). Consequently, both the software test research community and the en-terprise environment are working in developing techniques, mecha-nisms and tools that enable reducing test phase cost ensuring final quality results (Nazir and Khan, 2012).

Model-Driven Engineering (MDE) could be a solution to get this goal. This new software paradigm is based on the design and use of models to obtain software artifacts. Its application in test context is known as Model-Driven Testing (MDT), which is being successfully utilized in different Software Testing contexts (Völter et al., 2013). This paper presents an approach that, applied to the enterprise con-text, endorses this sentence.

This work also introduces a MDT approach that mainly focuses on a very concrete type of test: functional testing. This kind of testing tends to guarantee that initial requirements defined by users are correctly supported by the final system (Bertolino et al., 2005). This approach takes advantage of MDT power to define a set of models, transformations and processes that allows defining, in a systematic way, functional tests from functional requirements, reducing time and assuring the right traceability between initial requirements and final tests. The approach is authorized by the current real validation that we are getting in the enterprise environment.

This MDT approach comprises five metamodels and four transformations. Two metamodels are used to model the required information from functional requirements. The first transformation allows improving the requirement metamodel with a simple graph describing the functionality expressed in the requirement. Such graph is redundant as it does not describe any new information, but it helps simplifying other transformations. Path analysis and Category-Partition Methods are two of the main techniques used to derive test cases from functional requirements. Therefore, two metamodels and transformations have been designed to perform these techniques. Finally this MDT approach adds a fifth metamodel and a transformation to offer a consolidate view of the paths and categories.

From the authors' point of view, one of the main challenges for functional test case generation is the lack of formalism in functional test cases. With regard to advantages, functional requirements provide flexibility in the techniques, for example, by means of use cases or user stories. However, this flexibility is a gap to be filled in when trying to automate operations from use cases. The formalism of functional requirements constitutes a hard task in terms of time, knowledge and money. However, as this paper illustrates in the practical cases, once achieved, it provides a valuable return of investment.

This study is organized as follows: Section 2 presents the background that lists the terminology used in order to introduce the reader to the concrete context of the paper. Next, Section 3 offers a brief summary regarding related work on functional test cases and the most used testing techniques. Section 4 explains the approach by introducing the process of functional test cases generation from functional requirements under a MDT perspective in the first subsection; the second one analyzes in detail the set of used metamodels; and the third one defines transformations. Then, Section 5 details how the approach is implemented and provides the reader with a concrete example and practical references of the approach. To conclude, Sections 6 and 7 state the original contribution of this work together with the relevant conclusions and ongoing work.

2. Model-Driven engineering

As mentioned before, this paper focuses on the application of MDE in functional test cases generation. This section provides the reader with the lexicon and tools used along the text.

Two important elements must be stuck out in MDE environment: ***metamodels*** and ***transformations***.

Metamodels normalize the information used for generating test cases. A metamodel defines the constructor and the relations with constraints allowing models design with a valid semantics. Metamodels enable combining different concrete syntaxes (as textual and graphical syntaxes exposed in the motivational example) using the same lexicon and manipulating the same semantic artifacts.

A ***transformation*** is a relation between elements in a source metamodel and elements in a target metamodel. Therefore, executing transformation helps build a group of elements in target models to conform to their metamodels, using the information from a set of elements in source models, which must also agree with their metamodels. In this case, the agreed point guarantees that the transformation can be performed. Transformation elements define a systematic process regardless of any tool or programming language.

In this paper, UML (Unified Model Languages) (Object Management Group, 2011) class diagrams define the metamodel presented in the next sections, while QVT (Query-View Transformation language) (Object Management Group, 2010) identifies transformations. We select both solutions, as they are well-known standards. UML is proposed by OMG (Object Management Group) and it is frequently used in MDE for defining metamodels. OMG also suggests that QVT should be the standard for specifying transformations among models. Even though other possibilities are available to run our work, like ATL (ATL Transformation Language, 2015), we prefer using QVT since it has provided us with successful results in preceding projects (García-García et al., 2013).

QVT defines two main syntaxes for defining transformations: ***QVT-Relational*** and ***QVT-Operational***. *The former* is a declarative language similar to SQL. *The latter* defines operators and control structures from classic imperative languages. We have selected QVT-Operational for this paper because some of the transformations outlined in the subsequent section use a pathfinder algorithm.

A transformation in QVT is decomposed into a set of ***mapping operations***. A ***mapping*** is a relation between one or more source elements and one or more target elements.

Other elements used for defining transformations in QVT are ***queries*** and ***helper***. Both elements are operations that perform a computation and provide a result. A *helper* may have side effects on the given parameters, whereas a *query* has no side effects.

This paper also introduces the concept of ***direct mapping***. It defines a relation between one source element and one target element and a relation 1:1 between the attributes of the source element and the target element. Attributes from source and target elements have the same names and types. From this perspective, generating test cases from functional requirements becomes a problem when defining metamodels for functional requirements and test cases, and creating a set of transformations to get concrete test cases from particular functional requirements. Metamodels and transformations needed for producing functional test cases from functional requirements are presented in the next sections.

3. Related work

As the contextualization of our problem concerns, this section presents related work and it is divided in two parts. Section 3.1 summarizes the bibliography related to functional test cases generation from functional requirements. Then, Section 3.2 describes the two main techniques found in the literature previously studied.

3.1. State-of-the-art

At the time of writing this paper, there are two main surveys, (Escalona et al., 2011) and (Denger and Mora, 2003), studying existing approaches dealing with generating functional test cases from functional requirements. This section summarizes their most relevant conclusions.

These two considered surveys are specific for functional test cases, although other relevant comparative studies in this sense were published, such as Anand et al. (2013), where prominent techniques for automatic generation of software test cases are compared.

Conclusions from Denger and Medina's survey point out that the authors of the approaches do not follow any standards when defining templates (for functional requirements). On the contrary, each approach uses its own templates and formats. Another conclusion from that report is that none of the approaches uses path analysis techniques and, as a previous step, the approaches build a more formal representation of functional requirements.

Escalona's survey, developed by the same authors who write this article, concludes like the previous one. They mainly agree that many of the existing approaches have to formalize requirements as a first step to generate functional test cases, because of the use of text templates and colloquial language to define functional requirements. This is a mandatory step in approaches that offer a high degree of systematization and demand supporting tools. However, it can be pointed out that some approaches offer a systematic way, or even automatic ways to generate more formal models to automate the process. In this case, this is possible because requirements are described in natural language, therefore, they are metamodels and some transformations from this description enable translating requirements in natural language into activity diagrams.

Escalona's survey, published at the end of 2011, cites 24 approaches; the oldest dates back to 1988 (Category-Partition Method) and the newest to 2009. Denger's, published in 2003, cites 12 approaches; the oldest from 1988 (it is the same approach used in Escalona's survey) and the newest from 2002.

Below, there are some examples of the approaches included in Denger's and Escalona's surveys and new approaches not included in any surveys in order to update the current situation.

Hartmann et al. (2004) start their approach with functional requirements written in natural language. The result is a set of functional test cases obtained from a coverage criterion based on combinations that support Boolean propositions.

Binder's book (Binder, 2000) describes the application of the Category-Partition Method to use cases. Categories are any point in which the behavior of the use case may be different in two realizations of the use case. This application is named Extended Use Case Pattern.

In addition, Ibrahim et al. (2007) offer a tool, called GenTCase, which generates test cases automatically from a use case diagram enriched with every use case tabular text description.

Fröhlich and Link (2000) propose an approach describing how to translate a functional requirement from natural language into a statechart diagram in a systematic way, as well as how to generate a set of functional test cases from that diagram.

Ahlowalia (2002) presents an approach dealing with translating a functional requirement from natural language into a flow diagram and performing a path coverage technique to generate test cases.

Mogyorodi's (2003) approach describes functional requirements as cause–effect graphs that produce test cases from diagrams. Boddu et al. (2004) present an approach divided into two blocks: the first one presents a natural language analyzer generating a state machine from functional requirements, and the second one shows how to create test cases from such state machine.

Swain et al. (2010) introduce a similar approach to the one in Briand and Labiche (2002). First, a UML activity diagram is built with execution dependencies among use cases. This diagram indicates which use cases must be firstly executed. UML sequence diagrams define use cases and a coverage criterion is applied to extract execution scenarios from these diagrams. Test aces are the Cartesian combination of the path from the activity diagram and the scenarios from the sequence diagrams.

Sharma and Singh (2013) present an ongoing work based on an algorithm for deriving test cases from use case template, class diagram and data dictionary. No information about this algorithm is provided in this paper.

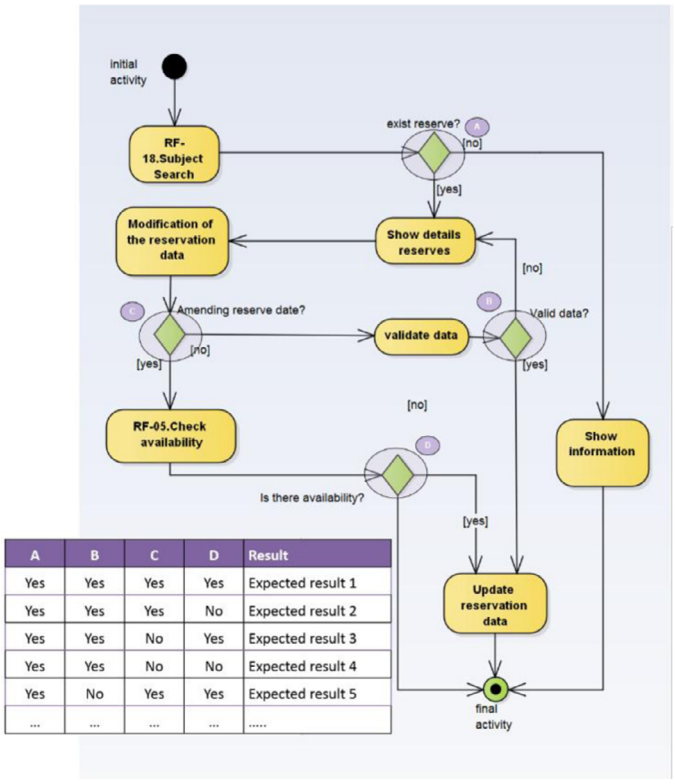
A recent paper (Nogueira et al., 2014) offers a strategy for the automatic generation of test cases of parameterized use cases templates. This approach considers a natural language representation that mixes control and state representation, which can be used to select particular scenarios during test generation. Unfortunately, it only covers sequential features, although it is conceived to cope with alternative features as future work.

The state-of-the-art introduced in this section indicates that no definitive approach closes the problem of generating functional text cases automatically in a satisfactory way. Thus, there are some aspects that may be improved such as the use of standards for inputs and outputs, the application of standards and more formal methods to describe the process itself, the need for empirical results, the measurement of possible automation and the need for a profitable supporting tool, among others. Some of these points are faced with MDT approach, as mentioned in this paper. All the cited approaches run with a concrete representation of functional requirements (as text templates, graphical diagrams or a mix of them) instead of defining the relevant information on functional requirements. As we present in Section 4.2.1, our approach considers these relevant elements. Therefore, they allow users to select the most adequate representation for their requirements.

3.2. Techniques for generating test cases

Our study in the previous section concludes that all approaches use one of the two (or both at the same time) testing techniques: Round-Strip Strategy and/or Extended Use Cases (terminology identified by Binder (2000)). Below, these techniques are described in depth (Fig. 1).

The Extended Use Case pattern consists in applying the Category-Partition Method (Ostrand and Balcer, 1988) to use cases. The Category-Partition Method is a technique based on identifying categories and partitions from the functional requirements under test and then, generating combinations among such partitions. These techniques guarantee a complete coverage of our functional requirements (Ali et al., 2014; Chimisliu and Wotawa, 2012). In this context, a category is any point for which the functional requirement defines an alternative behavior. Besides, a partition is defined as a domain subset of the condition evaluated in the category that decides whether a concrete piece of behavior is or not executed. Once all categories



A	B	C	D	Result
Yes	Yes	Yes	Yes	Expected result 1
Yes	Yes	Yes	No	Expected result 2
Yes	Yes	No	Yes	Expected result 3
Yes	Yes	No	No	Expected result 4
Yes	No	Yes	Yes	Expected result 5
...	...	...	...	.....

Fig. 1. Example of Extended Use Case test pattern.

and partitions are identified, they merge and each combination becomes a potential test case. The previous section presented several references about this topic in the specific context of use cases, like Hartmann et al. (2004) or Binder (2000). Fig. 2 shows an example of the Category-Partition Method (as described in Binder (2000)).

The Round-Strip Strategy deals with the application of a classic pathfinder algorithm to a state machine. Despite its concrete syntax, the behavior described in a functional requirement may be managed as a graph or state machine. Hence, a path searching allows identifying all the different paths through the behavior. Each path will be a scenario designed together with the system. Subsequently, each scenario is a potential test case for evaluating the right implementation of such scenario in the system under test. Generating test cases from state-machines is a widely referred topic in the research literature. Fig. 2 shows an example of the Round-Strip Strategy using a use case behavior defined as an activity diagram and the same behavior used in Fig. 2. Fig. 2 only represents three paths to illustrate this technique.

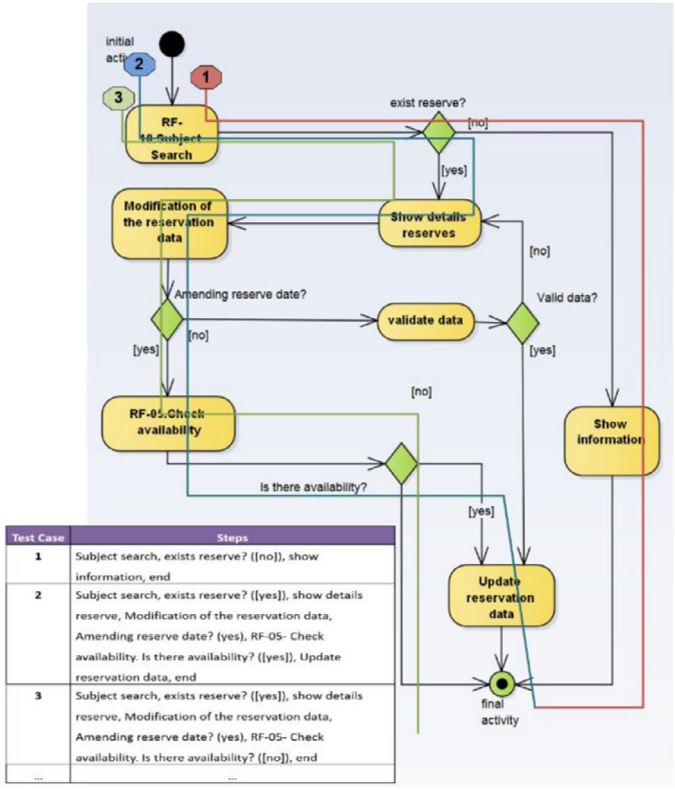
Next, Section 4 analyzes how our approach implements these two techniques for generating functional system test cases using metamodels and transformations.

4. A MDT approach for functional test generation

This section completely presents the approach. As it is a MDE approach, we introduce the two elements cited in Section 2: metamodels and transformations. However, as they can be very abstract concepts, if not visualized in a concrete context, this section firstly introduces a global view of the approach. Then, Section 4.1 describes the MDT process, Section 4.2 analyzes metamodels and Section 4.3 defines transformations.

4.1. The MDT process

This part offers a global view of test cases generation process from functional requirements, by means of metamodels and transformations, from a MDE point of view.



Test Case	Steps
1	Subject search, exists reserve? ([no]), show information, end
2	Subject search, exists reserve? ([yes]), show details reserve, Modification of the reservation data, Amending reserve date? (yes), RF-05- Check availability. Is there availability? ([yes]), Update reservation data, end
3	Subject search, exists reserve? ([yes]), show details reserve, Modification of the reservation data, Amending reserve date? (yes), RF-05- Check availability. Is there availability? ([no]), end
...	...

Fig. 2. Example of Round-Trip test pattern.

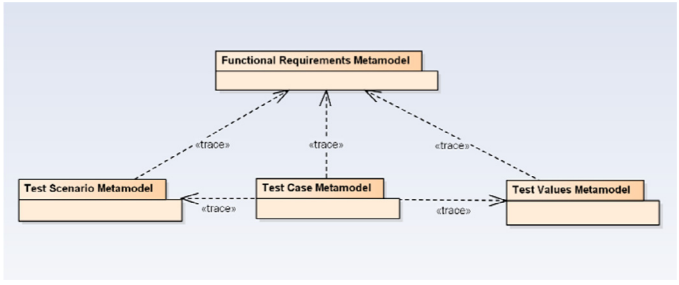


Fig. 3. Tracing relationships among metamodels.

Four metamodels have been analyzed (presented in the following section) for this approach. The first metamodel defines the concept of functional requirements. The second one enumerates the elements to model test scenarios according to the Round-Trip Strategy (see the previous section). The third metamodel lists the elements to model operational variables and the combination of operational variable values as established in the Category-Partition Method and the Extended Use Case pattern (see previous section). Finally, the fourth metamodel includes the elements to merge test scenarios, operational variables and combinations of values.

Fig. 3 shows the tracing relation among these four metamodels using a UML package model. Tracing enables knowing which test artifacts have been generated for each functional requirement.

An independency of the concrete syntax, notation or event tool in which the requirements are defined is allowed whenever the process focuses on metamodels. For instance, requirements may be modeled as text templates, state-machines, UML activity diagrams or by means of a proprietary tool. If functional requirements included the elements defined in the functional requirements metamodel, (further explained in the next section) the process described in this paper would be applied.

The same consideration may be given to the results. This work identifies the metamodel for testing artifacts, but it does not find any concrete notation. This means that textual templates or graphical notations may represent the generated artifacts and therefore, they are not attached to any specific tool.

The second part of the process concerns transformations. Four transformations have been found to generate functional test cases from functional requirements. They are depicted in Fig. 4 and briefly described in the next paragraphs.

During preliminary versions of these transformations, it was noticed that they worked with the scenarios of the requirement behavior or, using the metaphor again, all paths in FSM (Finite State Machine). Thus, all scenarios are created from a functional requirement by an individual and independent transformation T0 from Fig. 4, to avoid repeating codes and to reduce test cases complexity. These scenarios are entered as information related to functional requirements.

As our approach is based on the previous accepted techniques and our metamodels support elements of these techniques, we can guarantee that our generated tests are covered. In any case, when we implemented the solution in the tool, which is explained in Section 5, we tested these metamodels and transformations in order to check whether they produced the same result than the process carried out by hand.

Transformation T1 specifies how to obtain a test scenario model from a functional requirement model, if both models comply with the metamodels introduced in the prior section. Similarly, transformation T2 details how to obtain a set of categories and partitions from a functional requirement.

As it was analyzed before, the Test Case metamodel associates a test scenario with the related categories and partitions. Thus, transformation T3 uses both, test scenarios and test values models as input, and points out how to obtain test cases.

4.2. Metamodels

As Fig. 4 represents, four metamodels have been designed in our approach to define the information managed by the test cases generation process. These metamodels are described as follows.

4.2.1. Functional requirements metamodel

This metamodel does not aim to include all concepts of functional requirements, but only those relevant for generating test cases from use cases. We trust two different inspiration sources in order to detect which are these relevant elements. One of them is NDT (Navigational Development Techniques), which defines an own functional requirements metamodel based on the UML metamodel (concretely, behavior package) as it is presented in NDT (Navigational Development Techniques) (2015). The main advantage of this source is that it has been successfully applied in a large number of real projects. This fact guarantees its capacity to support functional requirements definition. A full explanation of NDT is out of the scope of this paper but several real examples and experiences of this methodology can be obtained in Escalona et al. (2007) and some details about the conceptual ideas of the approach are presented in Escalona and Aragón (2008). The other inspiration source concerns learned lessons from our comparative study, presented in Escalona et al. (2011) and presented in Section 3.1 of this paper. Fig. 5 proposes the metamodel with both sources and it is described along the next paragraphs.

The *Subsystem* element represents a package or container for functional requirements and other related elements (for example, *SystemActor*).

The *FunctionalRequirement* element is the key concept in this metamodel. The behavior of a functional requirement is modeled with two elements: *Step* and *ExecutionOrder*. The *Step* element models a concrete chunk of functional requirement behavior, such as requesting information or calculating a result. The *ExecutionOrder* element

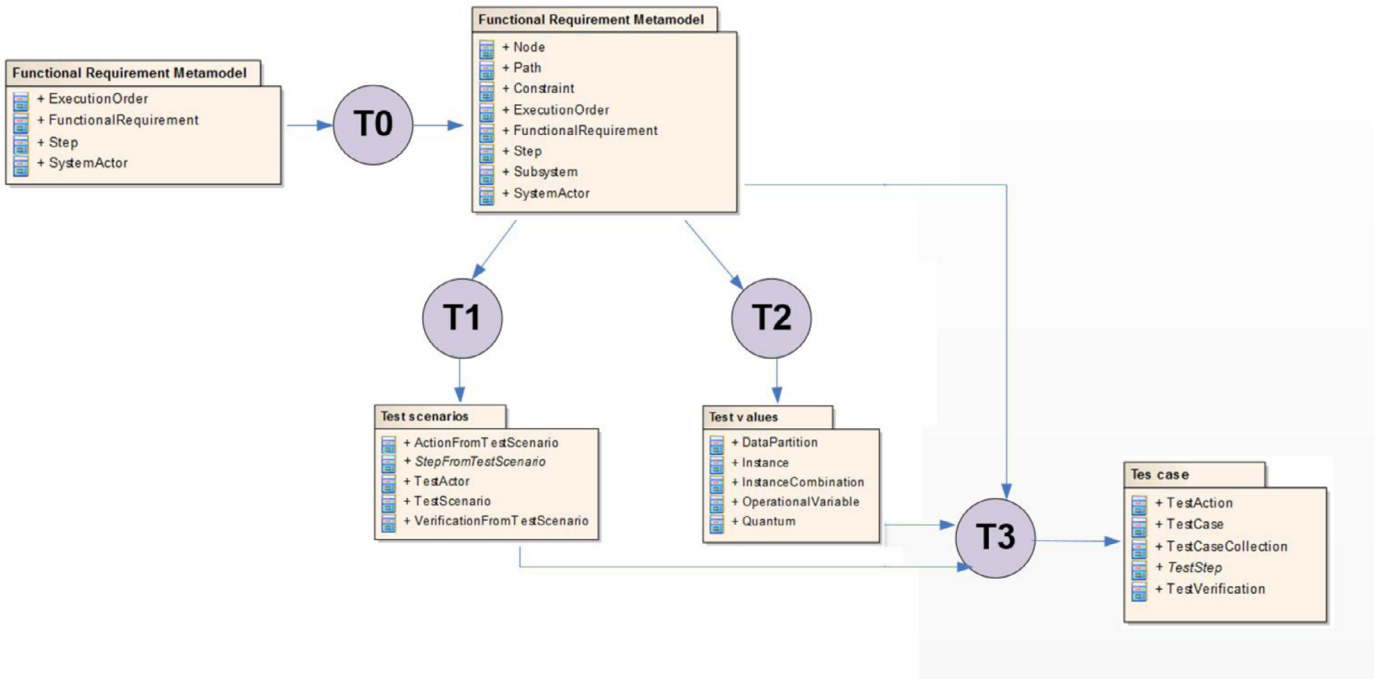


Fig. 4. Transformations and relations among metamodels.

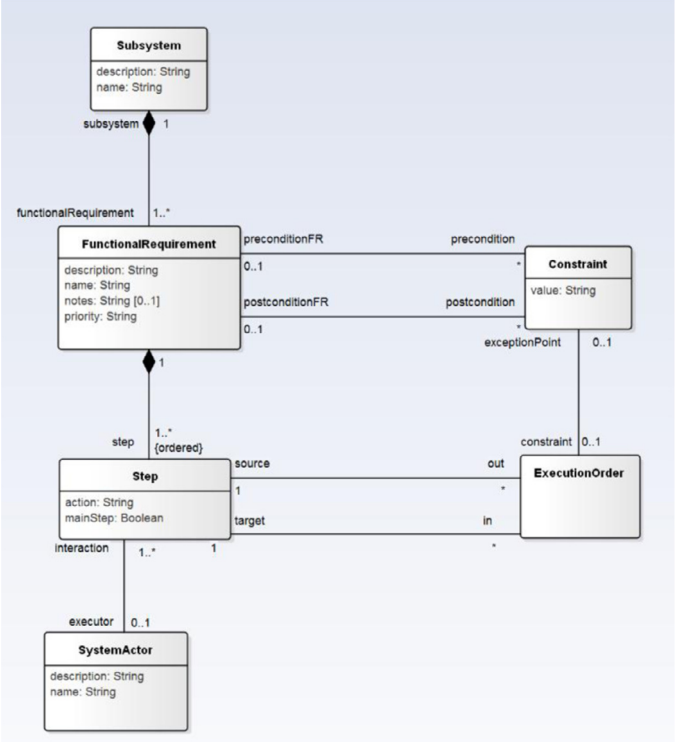


Fig. 5. Functional requirements metamodel.

defines the order to follow when executing the steps. The functional requirement may be seen as a FSM. The steps will be stated and the execution order will be transferred to a finite-state machine.

The *SystemActor* element models an external entity that interacts with the system, collaborating during steps performance.

Finally, the introduction of additional functional requirements as part of the functional requirement behavior has been taking into account by using the relation *reference-referencedFR* (from *Step* to *FunctionalRequirement*). This mechanism allows determining the semantics expressed by inclusion and extension relations, as UML Use Case metamodel (Object Management Group, 2011) defines.

The Constraint is a generic element that indicates Boolean statements. These statements are used to designate preconditions, postconditions and the extension condition of functional requirements.

4.2.2. Test scenario metamodel

The goal of this metamodel is to define the information obtained after applying the Round-Trip technique (Binder, 2000) (or other similar algorithm for path analysis in a state-machine). A test scenario is a possible functional test case for testing a concrete scenario from a functional requirement. The elements of this metamodel are represented in Fig. 6 and described below.

The *TestScenarioPackage* allows classifying test cases in the same way as the element package in UML or the element *Subsystem* in the previous metamodel do.

The key concept of this metamodel is the *TestScenario* element. A test scenario represents a concrete behavior of the tested functional requirement. Going back to the preceding metaphor, a test scenario will be a concrete path across FSM. The steps performed during the test scenario execution are classified in terms of the concepts defined below.

A *VerificationFromTestScenario* element models an action carried out by the system that may be verified in order to evaluate the test correction. Examples of possible verifications may be: amendments in the system data or other systems' or users' outputs.

An *ActionFromTestScenario* element is an action performed by an external element of the system under test. Examples of possible actions may be: a user's input or a server's response.

A *StepFromTestScenario* element is an abstract supertype for the actions in a test case. Instances of *StepFromTestScenario* element must be either a *VerificationFromTestScenario* or an *ActionFromTestScenario*

A *TestActor* element models any element participating in the test scenario and external to the tested system.

A *Constraint* element has been already introduced in the previous section.

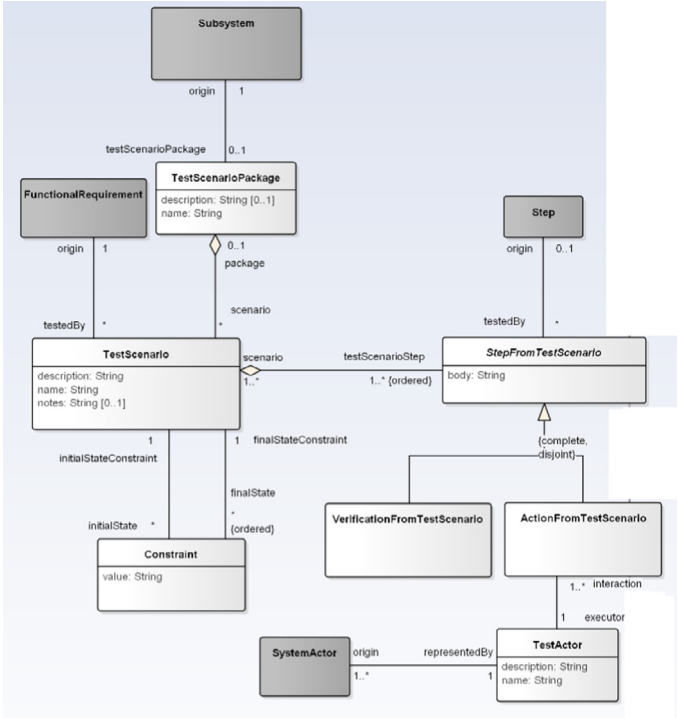


Fig. 6. Test scenario metamodel.

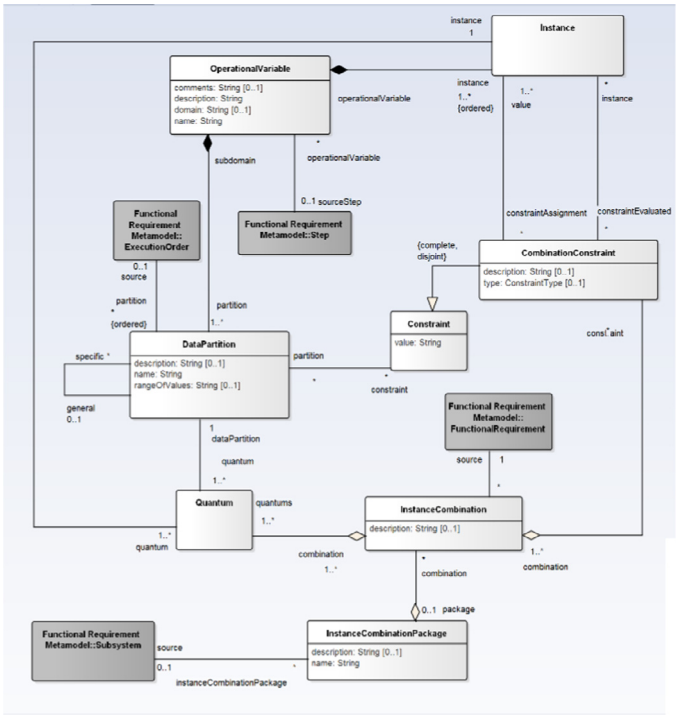


Fig. 7. Test value metamodel.

Darkness metaclasses (Fig. 6) are concepts from the functional requirement metamodel (in the preceding section) indicating traceability relations for the concepts in the test scenario metamodel. These relations allow recognizing the source functional requirement of test scenarios. Thus, at any time, it is possible to known the source functional requirement of a test scenario and the test scenarios testing a functional requirement.

4.2.3. Test value metamodel

The goal of this metamodel is to formalize the information obtained after applying the Category-Partition Method to functional requirements. Fig. 7 describes this model, which is further explained below.

The key concept of this metamodel is the *OperationalVariable* metaclass. This concept is the same as the concept of category, according to the Category-Partition Method.

The DataPartition element models each subdomain of the operational variable domain. During a test execution, an operational variable may take a value from one of its sets of partitions. However, some variables may take several values from different partitions during the same functional requirement, for example, if there is a loop in the behavior of this functional requirement.

For this reason, the elements *Instance* and *Quantum* are introduced in the metamodel. *Instance* models the participation of an operational variable in the functional requirement behavior and *Quantum* models the assignment between instance and partition from which instance takes the value in a concrete moment. For example, in a functional requirement that describes how the user may introduce a chunk of information and the system asks the information again, if it is wrong, an operational variable may be the information introduced by the user, and a test case may have two instances of that operational variable. Firstly, instance takes a value from the partition of invalid information and, secondly partition takes a value from partition of valid information.

The InstanceCombinationPackage element models a package to store instance combinations.

The Constraint element is introduced in Section 1.

In the classic Category-Partition Method, test cases are generated calculating the combinations between categories and partitions that confer a value to these categories. It is possible to model test cases as combinations of instances and quanta with previous elements.

The element *InstanceCombination* models a combination of instances. Nevertheless, if all combinations are calculated, some of them may be impossible to be executed in the tested system. For this reason, the element *CombinationConstraint* allows modeling a constraint in an "IF…THEN" format, in order to limit the produced combinations. The element *CombinationConstraint* enables introducing constraints to avoid specific combinations.

Again, darkness metaclasses in Fig. 7 are concepts from the functional requirement metamodel (in Section 1) indicating traceability relations for the concepts in the test scenario metamodel.

4.2.4. Test case metamodel

The last metamodel is the test case metamodel. It aims to combine the information and artifacts from test scenarios with the information and artifacts from test values. Thus, it is possible to manage test scenarios and test values in the same model. For this reason, this metamodel extends the elements from the test scenario metamodel to add relations with the elements of the test value metamodel (Fig. 8). Thus, using a test case is possible to know which combination of operation values is exercised and, how it takes a concrete value of an instance during the execution of one test step.

The key concepts in this metamodel are *TestCase* and *TestStep*. The *TestCase* element models a test scenario attached to the related combination of instances that the test case executes. In the same way, a *TestStep* is a test scenario attached to the quantum and the instance assigned. The elements defined in these metamodels have similar semantics to the elements of the test scenario metamodel.

A *TestStep* element can be either a *TestAction,* if it defines an interaction with an external actor, or a *TestVerification,* if the step designates a task performed by the tested system. That task could be used to validate the expected test case result.

A *TestCaseCollection* element is a test cases container.

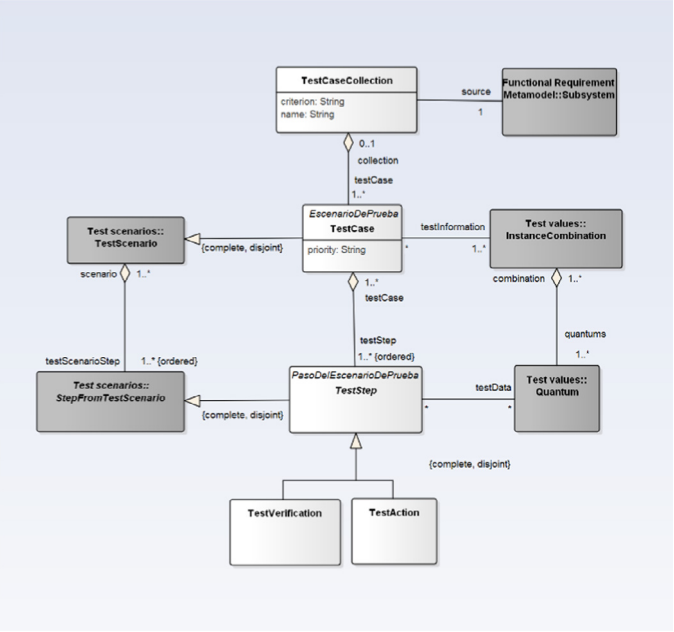


Fig. 8. Test case metamodel.

Darkness metaclasses in Fig. 8 are concepts from the functional requirement metamodel (in Section 1) indicating traceability relations of the concepts in the test scenario metamodel.

4.3. Transformations

The metamodels previously presented have depicted the information of functional requirements and functional test artifacts. The goal of this section is to define a process to obtain instances of the test metamodel from instances of the functional requirement metamodel. As it was mentioned at the beginning of this article, this process is modeled by means of transformations, which are relations oriented from a source toward a target metamodel, codified in QVT-Operational scripts. Fig. 4 represents a global view of transformations.

All transformations are specified in QVT and implemented in a supporting tool through Java language. This supporting tool is introduced in the next section and includes specific metrics to assure the quality of these transformations (Kapová et al., 2010). QVT specification has many low-level details, thus a high-level representation of the transformation process structure is explained in the following sections. The complete QVT code may be freely downloaded from IWT2 website (NDT (Navigational Development Techniques), 2015). This code is distributed under a BSD license (Open Source Initiative, 2014).

4.3.1. Transformation from functional requirements to (enrich) functional requirements

Before defining transformations, two new elements are added to the functional requirements metamodel in order to model test scenarios or paths from a functional requirement. These elements are *Path* and *Node*. A *Node* is a concrete step in a functional requirement. The main difference between a *step* and a *node* is that the former may have different inputs and outputs, whereas the latter has only one input (with the exception of the beginning node) and one output (with the exception of the ending node). Fig. 9 outlines the extension of the metamodel.

Both metamodels (functional requirements in Fig. 5 and the extended functional requirements models in Fig. 9) are elementary equivalent in that no additional information should be provided to create an instance of both metamodels and all transformations presented in this and next sections may be carried out with an instance

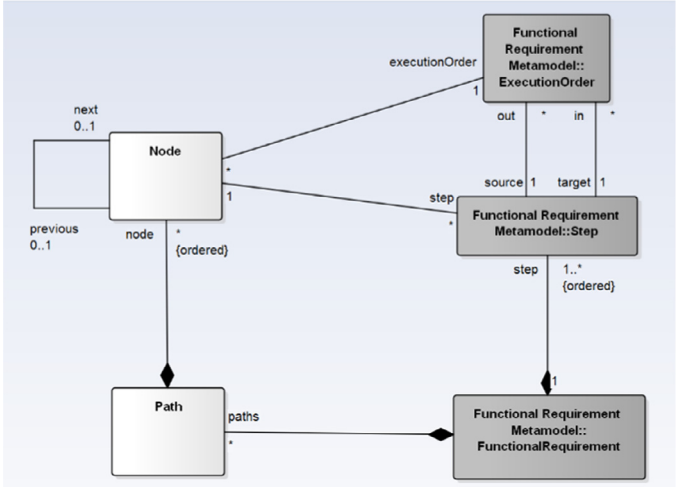


Fig. 9. Extension of functional requirements with scenarios.

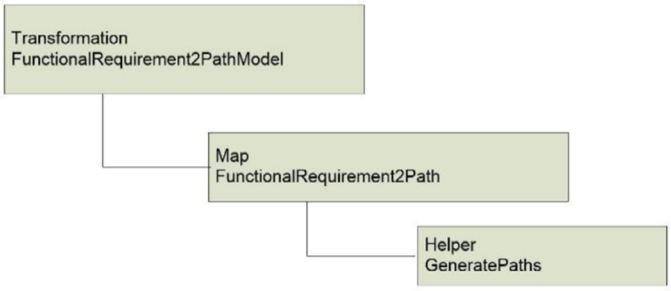


Fig. 10. Structure of T0 transformation.

of the two metamodels. The only difference is that we must execute some parts of the code several times, if we apply the transformation to a functional requirement metamodel instance (without *Path* or *Node* elements).

A transformation to generate these new elements is needed, after extending the functional requirements metamodel with two new elements. Therefore, such transformation aims to find all possible scenarios from a functional requirement behavior. As a result, it may be looked at as an implementation of a classic pathfinder or graph traversal algorithm using the QVT-Operational language. This transformation is considered as the implementation of a classic pathfinder algorithm by means of QVT Operational language. Fig. 10 offers an overview of this transformation.

Transformation in Fig. 10 invokes the mapping for every functional requirement, which in turn invokes the first call of the helper *GeneratePaths*. This helper is a recursive operation that traverses all the steps of a functional requirement, creating a set of paths with all the scenarios of the functional requirement. At the end of the generation process, all paths are added to the functional requirement.

From this point, transformations T1, T2 and T3 use the enriched functional model including all the scenarios as instances of *Path* and *Node elements*.

4.3.2. Transformation from functional requirements to test scenarios

The goal of this transformation (T1 from Fig. 4) is to obtain a model conformed to the test scenario metamodel presented in the previous section. Starting with the functional requirements enriched with paths generated by the transformation T0 (in the previous section), this transformation produces a test scenario for each path element.

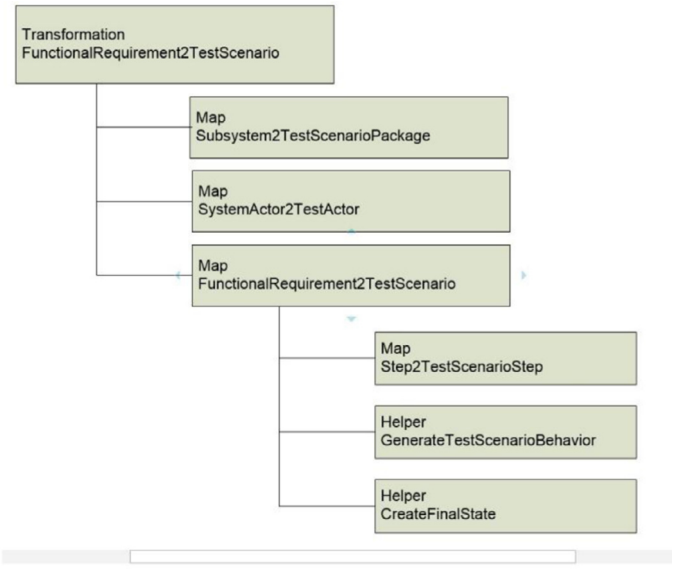


Fig. 11. Structure of T1 transformation.

```
transformation FunctonalRequirement_2_TestScenario (
    in  mrf: FunctionalRequirementMetamodel,
    out mep: TestScenarioMetamodel)
{
        main() {
            mrf.objectsOfType(Subsystem) ->
                map Subsystem_2_TestScenarioPackage();
            mrf.objectsOfType(SystemActor) ->
                map SystemActor_2_TestActor();
            mrf.objectsOfType(FunctionalRequirement) ->
                map FunctionalRequirement_2_TestScenario();

        }

        .....
}
```

Fig. 12. QVT code of transformation FunctionalRequirement2TestScenario.

```
mapping SystemActor::SystemActor_2_TestActor(): TestActor
{
    name := self.name;
    description := self.description;
    source := self;
}
```

Fig. 13. Mapping SystemActor2TestActor (example of direct mapping).

The transformation process firstly introduces the entry point. The main task of this entry point is to invoke three mappings depicted in Fig. 11. Fig. 12 shows the related QVT-Operational code.

The first direct mapping in Fig. 11 (Subsystem2TestScenario Packages) generates test scenario packages from subsystems directly. The second mapping (Fig. 11) also generates test actors from system actors directly. Fig. 13 shows the code for this mapping as an example of a direct mapping.

Finally, the third mapping generates test scenarios from functional requirements. This mapping is more complex than the previous ones, thus, it needs to call other additional mapping to create actions and verifications from functional requirement steps.

Mapping *Step 2TestScenarioStep* produces a test scenario from each functional requirement. This test scenario has no behavior yet. Then, the full set of test scenario steps appeared from all functional requirement steps. This mapping also classifies steps, specifying whether a verification action in the source step (of the functional requirement) has or not a relation with a system actor. Finally, in the third step, the created test scenario is cloned as many times as functional requirement scenarios are identified. A different scenario is selected for each test scenario cloned. The test steps for that scenario are identified and added to the test scenario.

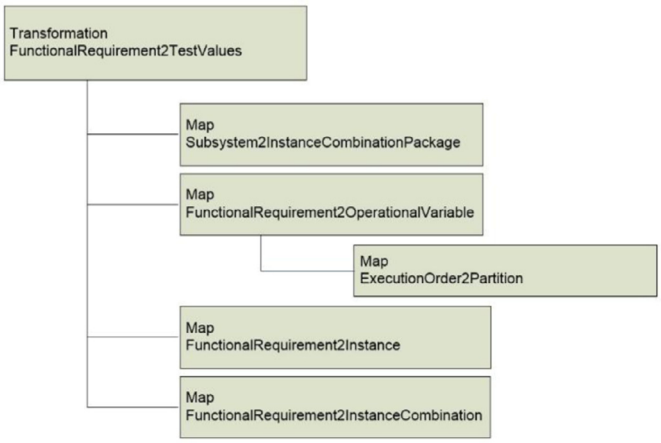


Fig. 14. Structure of transformation T2.

The helper *GenerateTestScenarioBehavior* clones test scenarios as many times as paths are generated from the functional requirement. The prior generated *TestScenarioStep* is located for each step and added to the final test scenario.

Finally, a test scenario is created for each functional requirement scenario, once the transformation has finished.

4.3.3. Transformation from functional requirements to test values

The goal of this transformation is to elaborate a model according to the test scenario metamodel presented in the previous section.

The transformation entry point just aims to call the four mappings in Fig. 14. Then, the direct mapping *Subsystem2InstancecombinatonPackage* generates packages for the combinations of instances replicating the same package structure of the functional requirements.

The second mapping, *FunctionalRequirement2OperationalVariable*, generates operational variables from a functional requirement. An operational variable is created for every point that offers more than one alternative to a functional requirement behavior. For example, if a functional requirement behavior defines a step that verifies whether a value is correct or not and, in the second case, the system asks for the value again, an operational variable is generated because in this step, the functional requirement may differ depending on the value.

Once the operational variable has been produced, this mapping invokes a second mapping (mapping *ExecutionOrder2Partition* from Fig. 14) to create partitions for the operational variable just generated. Each partition stems from each of the possible alternatives to the execution flow of the functional requirement behavior. For example, the operational variable produced in the previous paragraph will have two partitions; one will cope with all the correct values and the other one will cover all the wrong values.

The following mapping (*FunctionalRequirement2Instance* from Fig. 11) originates the instances of each operational variable created. As Section 4.3 explained (in the test value metamodel) an instance is a concrete evaluation or assignment of an operational variable that includes a value from one of its partitions. The same operational variable may be evaluated several times during a test, taking values for different partitions. This mapping traverses the set of paths and counts the maximum number of times that each operational variable may be evaluated. The result of that count is the number of instances generated for each operational variable.

Finally, the last mapping, *FunctionalRequirement2Instance Combination*, reveals the combinations of the instances created

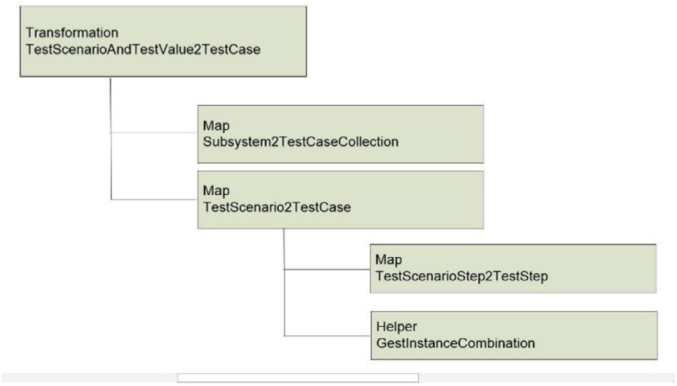


Fig. 15. Structure of T3 transformation.

in the previous mapping. The calculation of all combinations among instances and partitions may develop several combinations impossible to execute in the tested system. For example, if one instance takes a value from a partition that ends the functional requirement, then to keep on delivering combinations will be a waste of time. For this reason, only generated paths are used to identify valid combinations.

4.3.4. Transformation from test scenarios and test values to test cases

The goal of this transformation is to obtain a model that merges the elements of the two prior metamodels relating test scenarios to instance combinations (Fig. 15). Hence, this transformation takes both metamodels, a test scenario metamodel and a test value metamodel as inputs, and generates a test case model as output by merging the information of the input models.

The first mapping (*Subsystem2TestCaseCollection*) creates a folder structure to contain the test cases of the functional requirement model subsystem. Therefore, three test models (test scenario model, test value model and test case model) have the same folding structure, which improves the management tasks of test artifact and tracing relations.

Then, the *TestScenario2TestCase* mapping generates test cases for each test scenario directly. It invokes the mapping for generating test steps for the test scenario step of the test scenario, keeping the classification between verification and action steps. Then, it makes the helper search and recover the instance combination for that concrete scenario. Then, the mapping associates the combination of the test case and each step with the quantum, if any.

5. Implementation

Last section presents the theoretical basis of our approach. Now, this section offers the practical view. For this purpose, it is divided in four subsections. The first one justifies some implementation decisions. It not only aims to implement QVT or select a concrete syntax for metamodels, but it analyzes the use of Java to implement transformations and UML Profiles as concrete syntax for our approach. The second subsection offers an overview of a methodological Model-Driven approach and its tools, where our MDT approach was included. The third subsection explains how the MDT approach was included in the methodology and it is illustrated with a simple example, which is fully available on the web. This part finishes with some references to real published examples dealing with the experience in the enterprise context, which helps validate our approach.

5.1. Implementation decisions

As referred, the transformations introduced in the preceding sections have been formalized in the QVT-Operational code and they can be freely downloaded on the website. Table 1 compiles the QVT code metrics in order to widely present the tool dimension. [1] The result of comparing Java and QVT-Operational tools was considered a positive experience, although this comparison is out of the scope of this work. However, after analyzing some suitable tools, like SmartQVT (Telecom, 2007) and Moment (Boronat, 2007), we decide to use Java because these tools do not support full QVT. Implementing metamodels and transformations in Java code was an easy task thanks to testing, logging and debugging tools, together with a good IDE (Integrated Development Environment). On the one hand, the hardest part was to implement the ad-hoc code to generate functional requirements models from concrete syntaxes and, on the other hand, to generate concrete syntaxes from testing models.

Table 1
Metrics for QVT-Operational code.

	T0	T1	T2	T3	Total
Total lines	124	118	290	170	702
Lines of codes	104	97	238	124	563
No. of mappings	1	4	5	3	13
No. of helpers	1	2	3	1	7
No. of queries	3	2	1	3	9
No. of input models	1	1	1	3	6
No. of output models	1	1	1	1	4

5.2. A global view of NDT

We designed a tool that enabled putting this MDT process in practice, in order to test the suitability of our approach. After that, we framed it in a MDE framework, named NDT (Navigational Development Techniques) (Escalona and Aragón, 2008).

This paper will not present NDT in detail, as further information, videos and examples are available on IWT2 website to consult. Nevertheless, this section presents an overview so as to understand how this approach was included in the methodology environment.

Initially, NDT was defined as a MDE methodology focused on requirements and analysis processing. At the beginning, it dealt with defining a set of formal metamodels for the Requirements and Analysis phases. In addition, NDT identified a set of derivation rules, stated with the standard QVT, which generated analysis models from requirements models. The main goal of the NDT Requirements phase is to build the catalog of requirements containing the needs of the system to be developed. It is divided into a series of activities: capture, definition and validation of requirements. Requirements can also be classified according to their nature: information storage requirements, functional requirements, actor requirements, interaction requirements and non-functional requirements. NDT defines derivation rules to generate the analysis phase models, once the requirements specification phase has been completed and the catalog of system requirements has been drafted and validated. Fig. 16 shows this idea through the stereotype "*QVTTransformation*". The transition between the requirements and the analysis model is standardized and automated. It is based on QVT transformations, which translate the concepts of requirement metamodels into the first versions of the analysis models. These models are known in NDT as basic models of analysis. For example, the basic conceptual model of analysis is obtained from the storage requirements defined during the requirements phase.

Thereafter, the team of analysts can transform these basic models to implement and complete the final model of analysis.

The expertise of an analyst is required as this process is not automatic. Transformations are represented in Fig.16 through the stereotype "*NDTSupport*". NDT controls these transformations by means of a set of defined rules and heuristics to ensure consistency between

[1] We followed metrics suggested by (Ali et al., 2014).

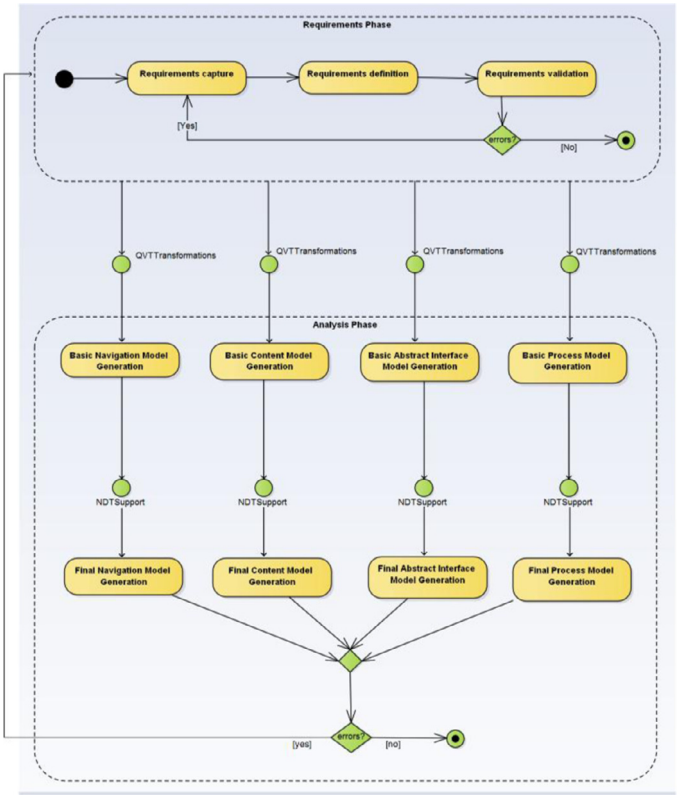


Fig. 16. Transformations from requirements to analysis.

requirements and analysis models. This idea, which is only applied in the Requirements and Analysis phases, was extended in the last years. Today, this context supports the complete life cycle (viability study, requirements, analysis, design, implementation, maintenance and testing), which is based on the approach presented in this paper. They run under different life cycles, such as classical, agile or the one based on prototypes.[2] To sum up, NDT offers an environment conducive to Web systems development, completely covering the software development life cycle.

However, the application of MDE and, particularly, the application of transformations among models, may become monotonous and very expensive, if there are no software tools that automate the process. Therefore, NDT has defined a set of supporting tools called NDT-Suite (García-García et al., 2012) to meet this need. Currently, the suite of NDT comprises the following main free Java tools:

(a) NDT-Profile is a specific profile for NDT, developed using Enterprise Architect. NDT-Profile offers the chance of having all the artifacts that define NDT easily and quickly, since they are integrated into a tool called Enterprise Architect (Enterprise Architect, 2015).
(b) NDT-Quality is a tool that automates most of the methodological review of a project developed with NDT-Profile. It checks both, the quality of using NDT methodology in each phase of software life cycle and the quality of traceability of MDE rules of NDT.
(c) NDT-Driver allows transformation can be automatically applied to every phase of the life cycle.
(d) NDT-Prototype is a tool designed to automatically generate a set of XHTML prototypes from the navigation models, described in the analysis phase, of a project developed with NDT-Profile.
(e) NDT-Merge uses the comparison among metamodels to identify syntactic and semantic inconsistencies among different versions of the same requirements catalog. In addition, NDT-Suite has more tools: NDT-Report, NDT-Glossary, NDT-Checker or NDT-Counter. The purpose of these tools can be verified on IWT2 website.

Nevertheless, the possibility of integrating our approach in a real methodological context with true practical experiences allows us to test the approach with authentic information.

5.3. NDT as MBT

Following the same ideas as NDT, we developed one UML profile for each metamodel described in Section 3.2. These profiles were included in NDT-Profile and transformations written in JAVA were implemented in NDT-Driver, the specific tool of NDT-Suite for transformations executions.

Fig. 17 shows an example of the behavior of a use case modeled using NDT-Profile, as well as the automatic result obtained after applying transformations to NDT-Driver. This example is based on a full theoretical example: the Hotel Ambassador.[3] In Fig. 17a, let us observe the description of use cases by means of an activity diagram. It describes how the user can make an online modification of his/her previous reservation at the Hotel Ambassador. The guest or the receptionist can start the use case looking for the room (executing another functional requirement in the system named FR-18 Subject Search). If there is a previous reservation, the system shows the results and enables the modification. After that, the guest or receptionist confirms data and the system checks availability through another functional requirement named FR-05 Check availability. If there is availability, the system updates the reservation data, if not, the process finishes without any change. The system also confirms previous reservations and then, the process finishes.

Fig. 17b shows the interface for NDT-Driver. As it was presented before, this tool supports QVT transformations, the original transformations of NDT (Feasibility Study to Requirements, Requirements to Analysis and Analysis to Design, which are out of the scope of this paper) and the implementation presented in Section 4.3. Thus, we select the project that defines the example in Fig. 17a in order to illustrate our case. Later, we can check Requirements-Testing checkbox and click "Transform" button.

In consequence, NDT-Driver executes the four transformations represented in Fig. 4.

Additionally, Fig. 17c represents a concrete functional test case as an activity diagram that is automatically obtained from the activity diagram Fig. 17a outlines. This test case is obtained through transformations and only represents one possible path from the activity diagram. It stands for the situation where there is a previous reservation, but any availability to go ahead with the modification. In fact, this example will generate a larger number of functional test cases, one for each possible path.

5.4. Real experiences

Space limitations do not allow introducing a full example. For this reason, the Hotel Ambassador includes video samples and a global example of our approach with more than 20 functional requirements that can be consulted by the reader.

In contrast, as we commented, this approach was fully applied on the enterprise context. In this section, we would like to mention three

[2] You can get more information about NDT full life cycle in www.iwt2.org.

[3] The example of NDT-Ambassador is fully available on www.iwt2.org. In fact, this website includes YouTube link to verify how this example runs. The full example with a large number of use cases can be freely downloaded.

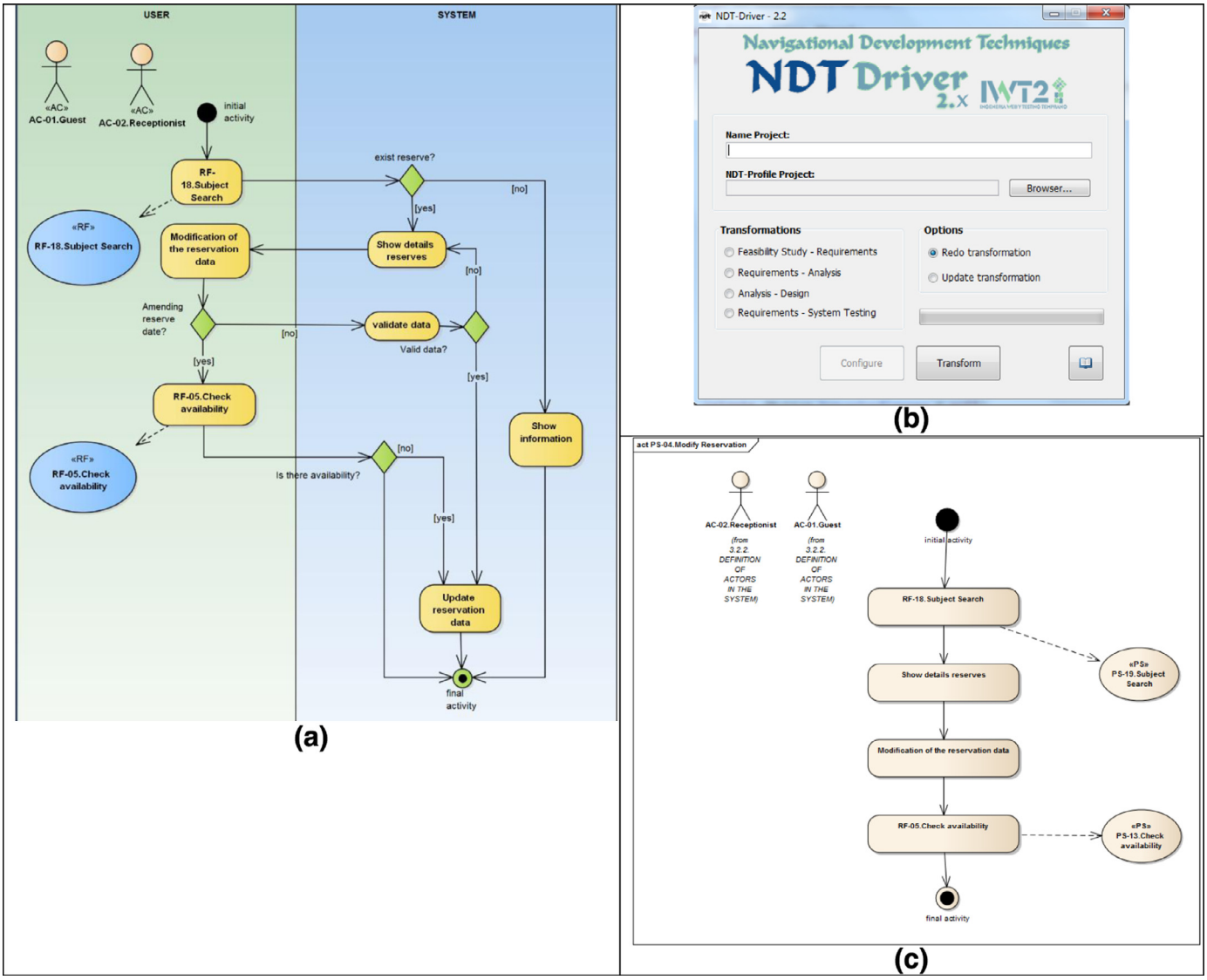


Fig. 17. An example of execution.

of them. We have selected three examples as the most representative, although NDT enriched with MDT approach was fully applied in last years. The first one, AQUA-WS, was the first project where it was used. It is worth highlighting that it inspired our approach. It was very interesting because it was developed by a combined development team (two different companies working together) with complex constraints of organization and very few resources for functional testing. It was developed with a classical life cycle.

The second one, CALIPSOneo, is a recent project that was developed with an iterative life cycle. It was a very illustrative experience for our team because it involved a very concrete, new and complex functional area. The last one concerned an e-government project developed in a public organization and it enabled us to value our approach in an agile context with a constant interaction with the final user.

Table 2 summarizes some of their objective characteristics, in order to illustrate the magnitude of these projects.

- AQUA-WS Project

The AQUA-WS Project was the first project where our approach was applied. The experience is fully described in Cutilla et al. (2012). Emasesa (2014) is a public company which deals with the general management of the urban water cycle, providing and ensuring water supply to all citizens in Seville, a city located in the south of Spain. In 2008, Emasesa decided to move its original systems to manage the water life cycle to a new exclusive and integrated Web system, which was named AQUA-WS.

The goal of this transfer was to develop an integrated management platform from scratch (but using a legacy database). The platform controls an integral business system for customer management and interventions in water distribution, cleaning and net management tasks. The software system is composed of three subsystems, each one representing a legacy system and additional generic subsystems to include all common elements (like database access), as well as allows exchanging information among the three subsystems. AQUA-RED is the subsystem responsible for managing the infrastructure of the pipe net; AQUA-SIC is the subsystem dealing with managing customers; and AQUA-SIGO is the subsystem in charge of managing the engineering projects, like construction or maintenance of infrastructures.

The new management system has 1808 functional requirements documented, all of them including several scenarios and alternatives.

Fig. 18 depicts the transfer process performed. First, the analyst team worked in the definition of the new system functional requirements. They were created by means of documenting the existing functionality unit of the legacy systems and validating them with

Table 2
Practical cases.

Project	Number of use cases	Number of test cases executed	Lifecycle	Total duration of the project
Aqua-WS	1873	1831	Classical	24
CALIPSOneo	77	139	Prototype	12
Thot	147	102	Agile	18

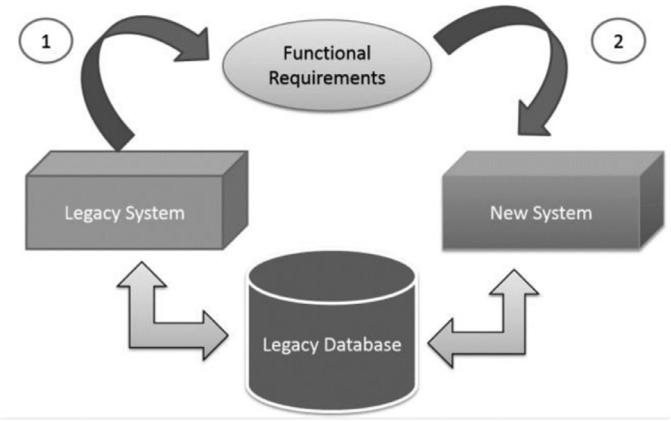


Fig. 18. Development process on AQUA-WS.

the expert users group. Then, the functional requirements were implemented from the scratch in the new system. The legacy database was developed to the new system.

Functional requirements were defined in NDT-Profile through two different syntaxes: UML Activity Diagrams and text templates. Activity diagrams were modeling according to UML specifications. Similarly, Text templates were modeling in terms of the previous work on functional requirements developed by the IWT2 group.

The aim of system testing was to verify every scenario from functional requirements. The estimated time to generate the package structure and the test case for every scenario, design those test cases in NDT-Profile and trace them with the tested functional requirement was measureless. Estimating a time of 5 min to create a test scenario in the modeling tool, the amount of time needed to generate a test case for each scenario was 583 h (73 days working 8 h per week, only for generating test cases). It was too much time spent to carry out a task that is repeatable and systematic, thus a tool support was proposed.

During the development of AQUA-WS project, the working teams decided to test our MDT approach using a prototype for elaborating the test plan. This plan contained over 7000 test cases produced from the different scenarios of the use case in a few minutes, replicating the package structure of the functional requirements and adding tracing relations with the tested functional requirements.

The application of test case generation does not demand high performance hardware. Test cases for AQUA-WS project are generated in 3 or 4 s using a common desktop.

Use cases from AQUA-WS project do not describe complex behavior. Each use case includes no more than 12 main steps and 4 or 5 exception scenarios. This process is repeated once for each functional requirement and each use case is processed individually, so there is no need to store information from a use case to another.

Although the number of requirements is vast for human perception (thousands of use cases), it seems indeed an irrelevant number in computational terms. Thus, a loop with thousands of cycles that performs an almost-constant work with an almost-constant memory use implies a very little amount of work and may be executed on any desktop or laptop just in a few seconds.

Information input/output from external sources is a classic bottleneck in algorithm. In this case, the software solution works with a database and a solid and stable driver. Hence, information input/output is performed in a fast and optimized way.

The suitable feedback obtained after the application of the prototype software tool to generate test scenarios from test cases motivated us to create a generic and formal approach and integrate it in the existing lines of work of the group.

- CALIPSOneo project

After AQUA-WS our MDT approach was fully implemented in NDT. A very recent project where it was applied was CALIPSOneo (*advanCed Aeronautical soLutIons using Plm proceSses & tOols*). It is a project leaded by EADS Casa (AIRBUS, 2015) whose main goal is to identify a working methodology that may allow engineers to define, simulate, optimize and validate aeronautical assembly processes on a 3D environment, before being executed in a real assembly line. This takes place through an integral process of requirements recollection, and the customization and use of the existing PLM software available.

Some years ago, several analyses were conducted on PLM methodology, concluding that there are many advantages for Airbus Military (Mas et al., 2013; AIRBUS, 2015). The knowledge obtained from such analyses help CALIPSOneo team compile the different requirements for the project, the management plans for the requirements and requirements' track matrix to meet the needs achieved for the PLM business. Therefore, CALIPSOneo covers both, the design of a new PLM methodology, adjusted to a collaborative PLM solution, and the development of prototypes that satisfy this concept.

CALIPSOneo is divided into three sub-projects with the aim of managing effectively the necessary work to finish the project. This enables the project scope to be managed in a more efficient way, as the different crews are working on the needed tasks to run the project. The three sub-projects are: MARS (*autoMAted shop-flooR documentation updating System*), PROTEUS (*PROcess sTructure gEneration and USe*) and ELARA (*gEneraLization to assembly oriented authoring Augmented ReAlity*). Detailed information about each of them and the experience of using NDT and MDT approach in this project is available on (Salido et al., 2014). This paper demonstrates the improvement of the use of this approach in this context, where the number of resources for testing is less than 10% in the full life cycle. This limitation was established by the clients' team, but we offer 40% coverage for testing.

CALIPSOneo offered the possibility to check our approach in a very different context: aeronautics. One of the most important aimed lessons of this experience was to analyze the suitability of this approach to test software in different contexts. If we use abstract syntaxes suitable to use metamodels described in Section 3, transformations help us to obtain tests.

- THOT Project

The last experience we expect to address is the THOT project. This experience is fully described in Ponce et al. (2014). It was executed in collaboration with AOPJA (Agencia de Obra Pública de la Junta de Andalucía, 2015). Thus, THOT project can be defined as an e-government project to implement an ECM (Enterprise Content Management) system in the Public Administration of the Regional Government of Andalusia (Spain). This project aims to make a qualitative

leap covering different disciplines of research and innovation, such as document management policies, e-government policies, dissemination and integration policies in Web environments (aspects that are treated and profusely investigated by different groups and research fora, both national and international). This fact enables organizations to provide a common framework for document management and cover the need to have a comprehensive management to complement business processes from beginning to end. The project offers, with innovation and research jobs, a solution that not only permits organizations to manage documents intelligently, but also to distribute, maintain and custody them.

NDT was used for the development of THOT. An essential activity for this project was the validation with the user. The specific context of the project required the continue collaboration with the user. In fact, the life cycle of this project was based on Scrum (2015). This experience helped us validate our approach in agile context environments with a very active interaction with the final user.

NDT solution was not only applied in these experiences. In fact, we aimed to present the aforementioned approaches, because there are detailed published papers that can be referenced for analyzing the advantages of our MDT approach in empirical contexts (Escalona et al., 2007). However, it is difficult to assess how our approach increased the results of these projects since we only measured successful results, but not testing without these techniques. We are working in this line of experiments in companies, according to Panach et al. (2015), in order to demonstrate the strong points. Currently, we have defined some experiments with companies' expert in testing so as to evaluate the real improvement that applying or not the approach involves.

6. Discussion and future work

As a final discussion, we would like to stick out that every automatic approach to generate test cases from a use case needs some assumptions and imposes some constraints on functional requirements, their format and context. This section studies these assumptions and constraints in the particular approach analyzed in this paper.

One of the drawbacks is that requirements should be conceived as a set of interactions between the system under test and a group of external actors. All software systems are not suitable enough to incorporate this kind of requirements. For example, compilers, embedded systems, videogames or basic CRUD (Create, Read, Update and Delete) systems cannot define such requirements since they hardly ever interact with users.

However, one advantage is that they may be well defined using specific test templates, UML state diagrams or UML Activity diagrams. There is a vast research and experience using these formats to define functional requirements. This previous work makes the reuse of existing approaches a relevant point. Thus, the functional requirement metamodel has been designed to be as easily adaptable as any other requirements approach exemplified in the requirements explained in the last section.

Another weakness is the level of abstraction of the generated test. As the test is produced from functional requirements, test cases are described using functional requirements concepts. Consequently, there is an additional and non-trivial work to execute it (manually or automatically) and the quality of the test cases obtained is proportional to the quality of the functional requirements input.

The practice also depends on the requirements phase and tools. Organizations with mature requirement phases and good automation tools may implement this process more easily than organizations lacking them.

As mentioned before, only one criterion to generate test cases has been implemented in transformations. However, there is no limitation to improve or use other graph-traverse criteria. The convenience of following those criteria is out of the scope of this work, as it is a subject widely described in the existing literature, for instance in Li et al. (2012).

Table 3
Characterization.

Dimension	Value
Subject	SUT
Model redundancy level	Shared test and dev model
Model characteristics	Deterministic, untimed and discrete
Model paradigm	Transition-based notation (FSM)
Test selection criteria	Structural model coverage and data-coverage criteria
Technology	Graph search algorithm
On/offline	Offline test case generation

With regard to concrete syntax, there is neither a generic mechanism nor a transformation to create a functional model from a concrete representation. On the contrary, each specific syntax needs a specific process (and its later implementation in the software tool) to obtain the relevant information and generate models from that information. It is out of the scope of this work to develop a tool that can cover any syntax or possible tool, but this tool can be adapted to the projects in which it is involved. For this reason, today such a tool only supports functional requirements as UML Activity diagrams. Nevertheless, it is possible to extend it to manage other syntaxes from other tools.

These reflections let us conclude that this work has opened new research lines. One of them deals with test cases prioritization mechanisms, consisting in giving relevance to functional requirements and studying the automatic detection of redundant test cases. The practice experience confirms that it continues producing a large number of redundant test cases that test teams have to identify manually.

Another relevant aspect regards the generation of operational variables. If it is deemed necessary to execute test cases, the automatic generation of data must also improve. The operational variable generation is not included in this first fusion. In fact, the test team has to produce data manually, what implies a hard task in this respect.

Utting et al. (2012) introduce a taxonomy for model-based testing approaches based on seven dimensions. Next paragraphs define a characterization of the approach introduced in this paper that matches the taxonomy of that work. Table 3 represents an overview of this characterization, although it is out of the scope of this work to present a deep description of the taxonomy.

Subject is the name given to the first dimension. In this MDT approach, the subject is SUT itself instead of the possible behavior of the SUT environment.

The model redundancy level indicates several uses for models. In our approach, this level is test cases and code, as this approach derives test cases from functional requirements and those requirements will be implemented in the system through source code.

The model characteristics dimension relates to nondeterministic, the incorporation of timing issues and the continuous or event discrete nature of the model. A use case model is a deterministic model and, in our approach, it does not include time issues. The iteration described in use cases is discrete.

The fourth dimension is model paradigm. It refers to the notation used to describe the model. Use cases behavior is finite state-machines (FSM) regarding formal notation, thus transaction-based notation is the proper value for this dimension.

The test selection criteria dimension defines the facilities that are used to control test cases generation. The approach introduced in this paper uses two different criteria. The structural model coverage criteria are in use when test cases are derived from use cases states and transitions of use cases behavior. Data coverage criterion works when test cases are derived from use cases variation points.

The test generation technology dimension describes the technology used during test generation. However, our technique uses graph-search algorithm mainly.

The last dimension is the on-line or off-line test generation. On-line test generation may test generation reaction to actual outputs of the SUT and off-line test case generation derives test cases before they are running. In light of this, our approach is considered an off-line test case generation, since it may produce test cases even before the system is built.

Another future line of research consists in assessing the relation between this approach and the UML Testing Profile (Object Management Group, 2013). This profile contains a package of stereotypes for defining the system behavior and another package for describing the test data based on categories and data partitions concepts. We will try to map the concepts from these approaches with similar concepts (if any) in the UML Testing profile.

Finally, as Section 5 stated, we are working in experiments to validate the effective improvement of our approach.

7. Conclusions

This paper has introduced a Model-Driven process for generating test cases from functional requirements. Section 7.1 introduces the conclusions and the original contributions of this paper and Section 7.2 exposes the lessons learned from the three projects in Table 2 and uses them to define a set of key points for introducing model-based testing in organizations.

7.1. Conclusions and original contributions

The surveys from Section 3 claim that existing approaches for generating test cases from use cases are attached to a concrete functional requirement notation. It also confirms that the processes have a lack of systematization. Some papers describe too generic processes to be performed automatically and other approaches offer concrete tools to carry out test cases generation. Nevertheless, those tools are attached to concrete notations or environments with specific inputs and outputs. The approach presented in this paper solves both detected lacks. Metamodels formally define the information needed without containing any concrete representation of the information. Instances of the elements may be represented with textual templates, UML diagrams or any other specific syntax. In addition, defining the generation process by means of transformations and QVT allows understanding the process in a systematic way and, again, regardless of the concrete implementation. This implementation may be performed with the supporting tools in Section 5, a handmade process or even in a QVT execution environment.

In conclusion, our approach provides some key and original points. Firstly, it introduces a set of metamodels for defining the relevant information to manage in the process. No previous work (from the surveys cited) has presented a metamodel. These metamodels open the door to adapt any existing approach to the generation process described in this paper.

Secondly, another relevant contribution not included in previous works is the notation independence. Metamodels describe the relevant information, but they do not demand a concrete specific notation. Moreover, the process does not require a particular syntax, consequently, it allows flexibility to adapt the most adequate syntax. Our experience reveals that the use of activity diagrams properly balances flexibility to define requirements and formality to apply an automatic process.

These metamodels are also the key to introduce another relevant contribution in the form of QVT transformations to generate test cases. We have introduced a tool to perform this process in a fully automatic way, as a side effect of this adding.

Finally, one of our strong points consists in applying the real area of our approach, as it was included in the context of NDT methodology and its tools. In this respect, it is worth noticing that it is actually a well-applied MDE methodology in the business environment. The paper shows three examples but, in the last years, a large number of projects in different contexts, with different functional environments, with different development and user teams and with different life cycle have utilized our approach successfully.

7.2. Lessons learned and key points

This section exposes the lessons learned after executing the present approach in the three projects represented in Table 2. These lessons are summarized as key points at the end of this section. The IWT2 group did not collaborate in those projects only for testing purpose. In fact, the real work of the group along the projects dealt with quality assurance. It mainly included requirements quality and the commitment of standards like a proper use of UML.

Quality assurance allowed the introduction of a homogeneous notation for requirements, which were defined in a tool using this notation for quality purposes. It was possible to introduce test cases generation as an added value, once the hard work was done. This strategy drove to the acceptance of the testing approach. Test case generation and other benefits from the NDT approach that are not interesting for this work, were perceived as the reward for the hard work regarding specifying good requirements.

A second lesson that led us to the successfully application of the approach was our costumers support; both public organizations that supported the financial cost of the projects as well as organizations that used the generated systems. These organizations were different from the development teams due to these teams came from hired entities.

The final customers committed with the quality of the documentation and models. This support appeared, for example, when developers did not aim to fix UML errors and omissions in the system models, arguing that they did not have enough time.

This compromise and support allowed a true formal and homogeneous requirement and a successfully generation of test cases. They allowed us to maintain quality even when the developer team claimed that they were not able to fix requirements because of time issues.

That support made the quality of requirements and, therefore, the possibility of generating test cases from them in an automatic way using this approach, real and true.

The third and final learned lesson was the role the authors and other members of the research group played as real testers of the systems. We were neither expert nor real users, but we had to interact with the system as users, trying to detect mistakes and flaws in the implementation. We needed some kind of test script for guiding us to interact with the system. In this case, we were clients from the test cases generated, for we used them to interact with the system.

Finally, as the main conclusion for this section, we would like to state that there are key points that may be considered as useful guidelines in future projects involving model-based techniques. They can be summarized as follows:

- Focus your efforts in quality and treat all model-based artifacts as reward for maintaining the quality of the products.
- Gain support and commitment to organizations, stakeholders and management teams.
- Get involved as a customer and user of the generated artifacts.

Acknowledgment

This research has been supported by the MeGUS (TIN2013-46928-C3-3-R) of the Spanish Ministry of Economy and Competitiveness.

References

Agencia de Obra Pública de la Junta de Andalucía. Consejería de Fomento y Vivienda. www.aopandalucia.es/. Last visit 05/ 2015.

Ahlowalia, N., 2002. Testing from use cases using path analysis technique. In: International Conference on Software Testing Analysis & Review.

AIRBUS. http://www.airbus.com/. Last visit 05/2015.

Ali, M.A., Shaik, K., Kumar, S., 2014. Test case generation using UML state diagram and OCL expression. Int. J. Comput. Appl. 95 (12), 7–11.

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Haman, M., Harrold, M.J., McMinn, P., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Software 86 (8), 1978–2001.

ATL Transformation Language. www.eclipse.org/atl/. Last visit 05/2015.

Bertolino, A., Marchetti, E., Muccini, H., 2005. Introducing a reasonably complete and coherent approach for model-based testing. Electron. Notes Theor. Comput. Sci. 116, 85–97.

Binder, R.V., 2000. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional.

Boddu, R., Guo, L., Mukhopadhyay, S., Cukic, B., September 2004. RETNA: from requirements to testing in a natural way. In: Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004. IEEE, pp. 262–271.

Boronat, A., 2007. MOMENT: A Formal Framework for MOdel ManageMENT (Ph.D. thesis). Universitat Politenica de Valencia (UPV), Spain.

Briand, L., Labiche, Y., 2002. A UML-based approach to system testing. Software Syst. Model. 1 (1), 10–42.

Chimisliu, V., Wotawa, F., 2012. Category partition method and satisfiability modulo theories for test case generation. In: 2012 7th International Workshop on Automation of Software Test (AST). IEEE, pp. 64–70.

Cutilla, C.R., García-García, J.A., Gutiérrez, J.J., Domínguez-Mayo, P., Cuaresma, M.J.E., Rodríguez-Catalán, L., Mayo, F.J.D., 2012. Model-driven test engineering – a practical analysis in the AQUA-WS project. In: ICSOFT, pp. 111–119.

Denger, C.M.M., & Mora, M.M. (2003). Test case derived from requirement specifications. Fraunhofer IESE Report, Germany.

Emasesa. Empresa Metropolitana de Abastecimiento y Saneamiento de Aguas de Sevilla. http://www.emasesa.com/. Last visit 08/ 2014.

Enterprise Architect. www.sparxsystems.com/products/ea/. Las visit 05/ 2015.

Escalona, M.J, Aragón, G., 2008. NDT. A model-driven approach for web requirements. IEEE Trans. Software Eng. 34 (3), 377–390.

Escalona, M.J., Gutierrez, J.J., Mejías, M., Aragón, G., Ramos, I., Torres, J., Domínguez, F.J., 2011. An overview on test generation from functional requirements. J. Syst. Software 84 (8), 1379–1393.

Escalona, M.J., Gutierrez, J.J., Villadiego, D., León, A., Torres, J., 2007. Practical experiences in web engineering. Advances in Information Systems Development. Springer, USA, pp. 421–433.

Felderer, M., Ramler, R., 2014. Integrating risk-based testing in industrial test processes. Software Qual. J. 22 (3), 543–575.

Fröhlich, P., Link, J., 2000. Automated test case generation from dynamic models. In: ECOOP 2000—Object-Oriented Programming. Springer, Berlin, Heidelberg, pp. 472–491.

García-García, J.A., Cutilla, C.R., Escalona, M.J., Alba, M., Torres, J., 2013. NDT-Driver: a Java tool to support QVT transformations for NDT. Information Systems Development. Springer, New York, pp. 89–101.

García-García, J.A., Ortega, M.A., García-Borgoñon, L., Escalona, M.J., 2012. NDT-Suite: a model-based suite for the application of NDT. Web Engineering. Springer, Berlin, Heidelberg, pp. 469–472.

Hartmann, J., Vieira, M., Foster, H., Ruder, A., 2004. UML-based test generation and execution. In: Präsentation auf der TAV21 in Berlin.

Huda, M., Arya, Y.D.S., Khan, M.H., 2015. Testability quantification framework of object oriented software: a new perspective. Int. J. Adv. Res. Comput. Commun. Eng. 4 (1), 298–302.

Ibrahim, R., Saringat, M.Z., Ibrahim, N., Ismail, N., October 2007. An automatic tool for generating test cases from the system's requirements. In: 7th IEEE International Conference on Computer and Information Technology, 2007, CIT 2007. IEEE, pp. 861–866.

Kapová, L., Goldschmidt, T., Becker, S., Henss, J., 2010. Evaluating maintainability with code metrics for model-to-model transformations. Research into Practice–Reality and Gaps. Springer, Berlin, Heidelberg, pp. 151–166.

Li, N., Li, F., Offutt, J., 2012. Better algorithms to minimize the cost of test paths. In: Fifth International Conference on Software Testing, Verification and Validation. Montreal. Montreal.

Li, Q., Yang, Y., Li, M., Wang, Q., Boehm, B., Hu, C., 2010. Improving software testing process: feature prioritization to make winners of success-critical stakeholders. J. Software: Evol. Process 24, 783–801.

Mas, F., Rios, J., Menendez, J.L., Gomez, A., 2013. A process-oriented approach to modeling the conceptual design of aircraft assembly lines. Int. J. Adv. Manuf. Technol. 67 (1–4), 771–784.

Mogyorodi, G.E., 2003. What is requirements-based testing? Crosstalk. J. Defense Software Eng. 16.

Nazir, M., Khan, R.A., Testability Estimation Model (TEMOOD), January 2012. In: N., Meghanathan, N., Chaki, D., Nagamalai (Eds.), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 85. Springer-Verlag, pp. 178–187 Part 3.

NDT (Navigational Development Techniques) site. http://www.iwt2.org/web/opencms/IWT2/ndt/?locale=es. Last visit 05/ 2015.

Nogueira, S., Sampaio, A., Mota, A., 2014. Test generation from state based use case models. Formal Aspects Comput. 26 (3), 441–490.

Object Management Group. (2010) Query View Transformation Specification 1.0. http://www.omg.org. Last visit 05/2015.

Object Management Group. (2011). Unified Modeling Language 2.4. www.omg.org. Last visit 05/2015.

Object Management Group. (2013). UML Testing Profile 1.3. Last visit 05/2015.

Open Source Initiative. The BSD License. www.opensource.org/licenses/bsd-license.php. Last visit 08/ 2014.

Ostrand, T.J., Balcer, M.J., 1988. The category-partition method for specifying and generating functional tests. Commun. ACM 31 (6), 676–686.

Panach, J.I., España, S., Dieste, Ó., Pastor, Ó., Juristo, N., 2015. In search of evidence for model-driven development claims: an experiment on quality, effort, productivity and satisfaction. Inf. Software Technol. 62, 164–186.

Ponce, J., Dominguez-Mayo, F.J., Gutierrez, J.J., Escalona, M.J., 2014. Pruebas de aceptación orientadas al usuario: contexto ágil para un proyecto de gestión documental. Ibersid 8, 13–22.

Salido, A., García, J.A.G., Ponce, J., Gutierrez, J.J., 2014. Tests management in CALIPSOneo: a MDE solution. J. Software Eng. Appl. 7 (06), 506.

Scrum. https://www.scrum.org/. Last visit 05/ 2015.

Shah, H., Harrold, M.J., Sinha, S., 2014. Global software testing under deadline pressure: vendor-side experiences. Inf. Software Technol. 56 (1), 6–19.

Sharma, A., Singh, M., April 2013. Generation of automated test cases using UML modeling. Int. J. Eng. Res. Technol. 2 (4).

Swain, S.K., Mohapatra, D.P., Mall, R., 2010. Test case generation based on use case and sequence diagram. Int. J. Software Eng. 3 (2), 21–52.

Telecom, F. (2007). SmartQVT: an open source model transformation tool implementing the MOF 2.0. QVT-Operational language.

Utting, A., Prestschner, A., Legeard, B., August 2012. A taxonomy of model based testing. Software Test. Verification Reliab., 22, pp. 297–312.

Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., 2013. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.