Towards the Automated Generation of Abstract Test Cases from Requirements Models

Maria Fernanda Granda[1]
Department of Computer Science
University of Cuenca
Cuenca, Ecuador
fernanda.granda@ucuenca.edu.ec

Nelly Condori-Fernández
Department of Computer Science
VU University
Amsterdam, Netherlands
n.condori-fernandez@vu.nl

Tanja E.J. Vos, Oscar Pastor
[1]DSIC
Universitat Politècnica de València
Valencia, Spain
{tvos, opastor}@dsic.upv.es

***Abstract*—In a testing process, the design, selection, creation and execution of test cases is a very time-consuming and error-prone task when done manually, since suitable and effective test cases must be obtained from the requirements. This paper presents a model-driven testing approach for conceptual schemas that automatically generates a set of abstract test cases, from requirements models. In this way, tests and requirements are linked together to find defects as soon as possible, which can considerably reduce the risk of defects and project reworking. The authors propose a generation strategy which consists of: two meta-models, a set of transformations rules which are used to generate a Test Model, and the Abstract Test Cases from an existing approach to communication-oriented Requirements Engineering; and an algorithm based on Breadth-First Search. A practical application of our approach is included.**

***Index Terms*—Requirements-based testing, Communication Analysis, Model-driven testing, Conceptual Schema Testing, Test Model Generation, Test Case Generation.**

I. INTRODUCTION

Testing aims to detect defects in a system by comparing the expected results (expressed in system requirements) to the observed results (the behaviour of the implementation of the System Under Test (SUT). In order to detect defects before they become extremely expensive to fix and manage the inevitable changes during the software lifecycle, testing activities should start as soon as possible (the requirements level) in the software lifecycle.

In paradigms such as Testing-Driven Development (TDD) [1] and Behaviour-Driven Development (BDD -an evolution of TDD) [2], the tests are written in an incremental and iterative way prior to the production code as a specification of functional tests (e.g. TDD) or specifications of the product's behaviour (e.g. BDD). But the tests are executed for testing the SUT at code level.

Additionally, the Requirements-Based Testing [3] methodology has emerged as a solution and considerably reduces the causes of project failures, defects and reworking by addressing two major issues: (i) validating that the requirements are correct, complete, unambiguous and logically consistent; and (ii) designing a necessary and sufficient set of test cases from these requirements to ensure that the design and code fully meet these requirements.

However, in the testing process, the design, selection, creation and execution of test cases is a very time-consuming and error-prone task when done manually, because suitable and effective test cases must be obtained. As the automatic generation of test cases will reduce the cost of the testing process, increase the effectiveness of the tests and optimize the testing cycle [4], in this work we try to address the challenge of automatically generating test cases of sufficient quality by optimizing coverage and minimising testing costs.

Model-Driven Engineering (MDE) [5] advocates for the use of models as development artefacts, which can be applied for facilitating communication by hiding technical details, specifying its structure and behaviour in an understandable way, or even generating test cases. In this paradigm, the quality of the conceptual schemas (the system model) becomes a key factor that requires methodologies and procedures to assure that the conceptual schema meets the requirements specified.

This paper presents an approach that automatically generates a set of abstract test cases, for testing conceptual schemas, from the requirements models. For this purpose, we apply the Model-Driven Testing (MDT) paradigm [5], which means that it uses a models transformation strategy for generating test models. MDT has certain advantages, such as i) rules are specified once, then the same derivation can be re-used for generating test cases from multiple requirements models; and ii) platform independence, the executables test cases can be generated in different target codes.

The main contributions of the paper are the following:

- A model-driven testing approach to automatically generate abstract test cases from the requirements model, based on communication-oriented business process specifications.
- Artefacts (meta-models, transformation rules and algorithm to establish the different sequences of the test items) that are defined and implemented to support the MDT strategy.

The proposed approach is illustrated by means of an example.

In the following, we briefly summarize the basic testing concepts used in the study. Section III summarizes the Requirements Model used in the proposed approach. Section IV defines the meta-models and transformation rules used. Section V presents an overview of the generation process of the test model and the abstract test cases applied in the development of the practical application given as an example. Section VI describes related work. Finally, conclusions and future work are summarized in Section VII.

II. Basic Concepts

This section describes the concepts and testing artefacts used in the generation process of the test cases.

A. Testing Concepts

Following standard terminology [6], the following definitions are used in this paper.

A *test case* is a set of input values, execution pre-conditions, expected results and execution post-conditions.

An *abstract test case* is a test case without concrete (implementation level) values for input data and expected results.

A *concrete test case* is a test case with concrete (implementation level) values for input data and expected results.

B. Testing Artefacts Involved

The *requirement model* describes the system requirements at business level. The requirement model is specified by the domain experts and system analysts and is an instance of the Requirements Meta-model proposed by España [7].

A *conceptual schema* (CS) defines the general knowledge that an information system needs in order to perform its functions [8].

The *conceptual schema under test* (CSUT) is a conceptual schema in an executable form (e.g. Foundational subset for Executable UML Models –fUML [9] with Action Language for Foundational UML –Alf [10]).

The *test model* (TM) contains information about the test items and their order of precedence, which are generated from the requirements model. This model conforms to the Test Model Meta-model (TMM). The meta-model is discussed in Section IV.

The *abstract test cases* are obtained from the test Model. These are structured sequences of the test cases (e.g. services, triggers, assertions and links) for conceptual schemas exemplifying the interaction of actors with the system. The test cases are abstracts in the sense that they do not contain concrete objects.

The *concrete test cases* (test code) are generated from the abstract test cases with concrete data.

The *executable test cases* are the concrete test cases converted into executable script files. For this purpose, languages such as Alf can be used.

Since our proposal complies with the principles of Model-Driven Testing, it distinguishes different types of models at various levels of abstraction, such as those shown in Fig. 1, the Platform-Independent Test model (PIT) and the Platform-Specific Test model (PST).

Our MDT process requires three transformations: (i) the first one is a model to model transformation (M2M): from the requirements model (which is a Platform Independent Model - PIM) to the test model; (ii) the test model (PIT) is converted into an abstract test cases model (PST –M2M transformation); (iii) finally, the concrete and executable test cases will be generated into test script using a vertical transformation (PST to code – model to text transformation).

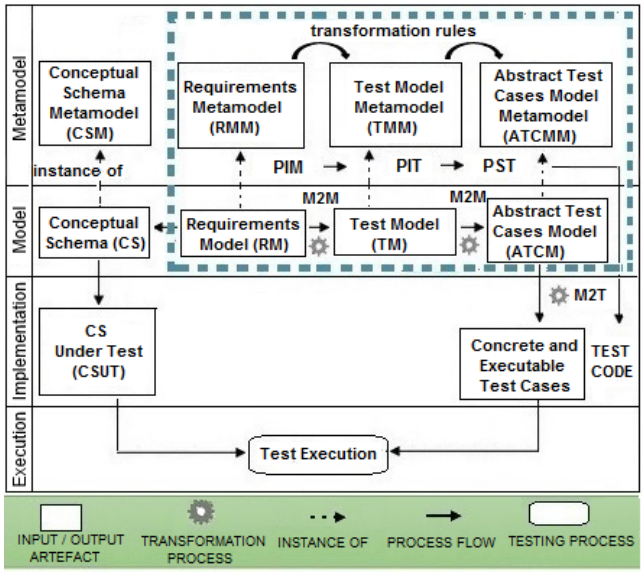

Fig. 1. An overview of our MDT approach from requirements models

The scope of this paper includes the first two transformations (see Fig 1).

In the next section, we identify the most relevant elements of the Requirements meta-model from a testing viewpoint.

III. Requirements Model: Identifying the Main Primitives from a Testing Viewpoint

Communication Analysis (CA) is a Requirements Engineering method which aims to analyse and structure requirements focuses on communicative interactions that occur between an enterprise Information Systems (IS) and its environment [11]. For this purpose, a requirements structure with five levels (i.e. system/subsystem, process, communicative interaction, usage environment; and operational environment) is proposed in this method [11].

Our proposal covers the requirements related to communicative events. They have been structured in three wide categories [11]: contact (requirements related to the triggering of the event by an actor to communicate something to the information system, e.g. preconditions), message (specify the contents of the message being communicated to or from the IS, e.g. message fields, domain of the message fields, message constraints); and reaction requirements (information system reaction, e.g. linked behaviours and linked communications).

Actually, Communication Analysis has been integrated into the OO-Method, an object-oriented Model-Driven Development framework [12] that is UML-compliant.

A model transformation strategy has been defined to derive, from Communication Analysis requirements models, initial versions of OO-Method conceptual models that can already be compiled to automatically generate software code [13], [7].

Among the techniques for requirements specification using CA, the Communicative Event Diagram and the Event Specification Templates are the main artefacts used in the present approach.

A. Communicative Event Diagram (CED)

CED is a graphical modelling technique to represent a business process model, where the notation is similar to the UML Activity Diagrams, but differs from the Activity diagram and other proposals (e.g. Business Process Modelling Notation -BPMN) in that it includes the primitives (Event Specification Template primitives) that a model-driven method needs (fine-grained enough to be represented directly in code) to express the structure and dynamics of an IS. Figure 2 shows the communicative event diagram of the business process of the Online Conference Review System (for further information on this system see Appendix I).

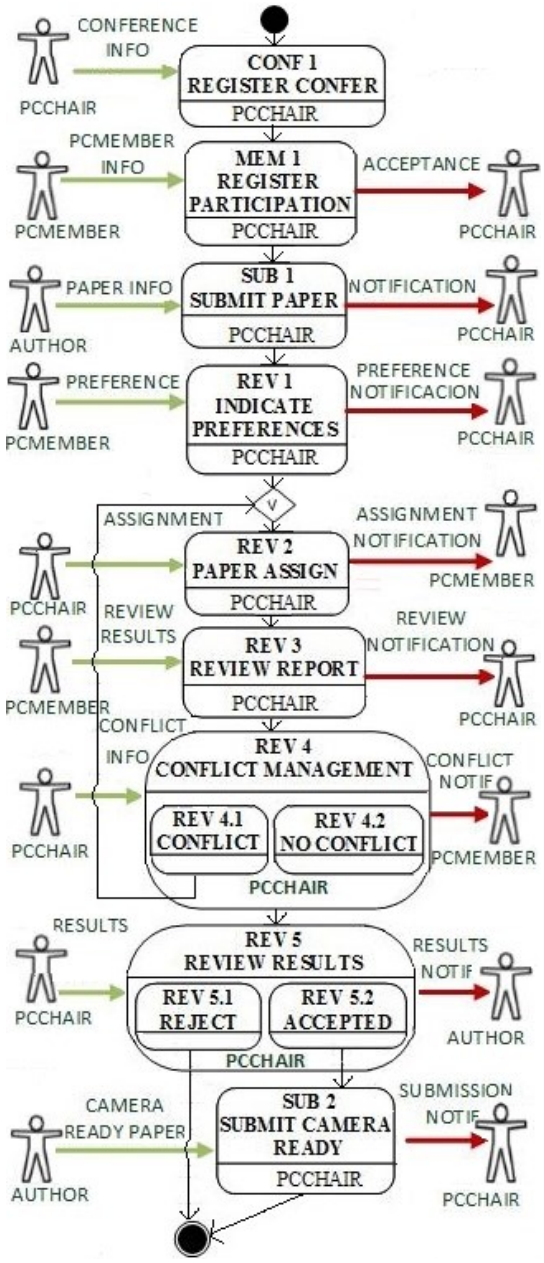

Fig. 2. OCR system requirement model based on a communicative event diagram

The following modelling primitives are of interest for the purpose of automatically deriving Test Models.

A *Communicative Event* is a set of actions related to information (acquisition, storage, processing, retrieval and / or distribution) carried out when an external stimulus occurs and have to be performed completely and uninterruptedly according to the unity criteria [11] (e.g. events CONF1, MEM1 and so on, in Fig. 2). For each communicative event a message containing meaningful information is transmitted to the IS.

An *event variant* (i.e. events 4.1, 4.2, 5.1 and 5.2 in Fig. 2) refers to each alternative behaviour within a *specialised communicative event* (i.e. events 4 and 5 in Fig. 2).

Even though actors are essential for the requirements model based on communication analysis, as yet they are not considered to play a significant role in the derivation of the Test Model for Conceptual Schemas, because they are only responsible for communicating the new meaningful information to the information system. However, this information is not complete enough to formulate requirements such as access control requirements.

A *precedence relation* between two communicative events A and B indicate that "A is a precedent communicative event of B"(see arrows between communicative events in Fig. 2). The temporal ordering of the CED communicative interactions facilitates obtaining the sequence of the test items in a systematic way.

Logical gate *Or*. The or-merge indicates that only one of the incoming precedence relations needs to hold (see diamond in Fig. 2).

Logical gate *And*. The and-fork and the and-join are implicitly represented by two or more precedence relations leaving from (or arriving at) a communicative event; however, they can be explicitly drawn if needed to express complex logical expressions.

A *start node* indicates the beginning of the CED and *end node* represents the end of the CED.

B. Event Specification Template (EST)

An event specification template is a textual specification technique that is used to describe, by means of a Message Structure, both ingoing and outgoing messages transmitted to the IS in a *Communicative Event* [14]. TABLE 1 shows the message structure for the communicative event SUB1 (an author submits a paper) in our example. The following grammatical constructs are of interest for the purpose of Test Model derivation (see [14] for further information on this technique).

A *Substructure* is an element that is part of a message structure. For example, VERSION, AUTHOR, TOPICS and CONFLICTS are substructures of SUBMISSION.

The initial substructure is the first level of a message structure. In our case SUBMISSION = <id Submission + Abstract + Keywords + Tittle + VERSIONS + AUTHORS + TOPICS + CONFLICTS>.

There exist two classes of substructures;

1) Field: Basic informational element of the message and is not composed of other elements.

TABLE 1. PARTIAL VIEW OF THE MESSAGE STRUCTURE FOR COMMUNICATIVE EVENT SUB1

Field	OP	Domain	Example Value
SUBMISSION =			
<id Submission	g	text	Submission10
Abstract +	i	text	There is an …
Keywords +	i	text	Test Cases, …
Title +	i	text	Towards …
VERSIONS =			
{ VERSION =			
< id File +	g	text	F0010-01
File +	i	text	Submis10.pdf
Type +	g	text	Submission
Date> }	i	text	10-04-2014
AUTHORS =			
{AUTHOR =			
< id Author +	g	text	A00345
Name +	i	text	Fernanda
Last Name +	i	text	Granda
Username +	i	text	fgranda
Password +	i	text	21212
Organization+	i	text	UPV
Country +	i	text	Spain
Email > }	i	text	fg@pr.upv.es
TOPICS =			
{ Topic }	i	Topic	T01,..
CONFLICTS=			
{ PCMEMBER } >	i	PCMember	None

a) Data Field: Piece of data with a basic domain[1]. For instance, Abstract, Title.

b) Reference Field: Field whose domain is a type of business object. For instance, Topic references is a topic that is already known to the IS.

2) Complex substructure: Any substructure that has an internal composition.

a) Aggregation Substructure: Specify the composition of several substructures in such a way that they remain grouped as a whole. It is represented by angle brackets < >. For instance, VERSION= <id File + File + Type + Date>.

b) Iteration Substructure: Specify a set or repetition of the substructures it contains. It is represented by curly brackets { }. For instance, a submission can be related to several VERSIONS, AUTHORS, TOPICS and CONFLICTS.

Each field is characterised by properties, some of which are described below.

It must have a significant *Name* (e.g. Abstract).

An acquisition *operation* (OP) specifies the origin of the information that the field represents.

- Input (i): The information of the field is provided by the primary actor.
- Generation (g): The IS can automatically generate the field information.
- Derivation (d): The field information is already known by the IS and therefore can be derived from its memory; i.e. it was previously communicated in a preceding communicative event. This operation can have an associated derivation formula.

If the attribute operation is of the "derivation" type, the *derivation formula* indicates the formula in natural language, or Object Constraint Language (OCL[2]).

A *Domain* specifies the type of information that the field contains.

An *Example Value* is a value for the field, provided by the organisation.

The *minimum Cardinality* is a value that indicates the minimum cardinality of the data field.

The *maximum Cardinality* is a value that indicates the maximum cardinality of the data field.

An *isIdentifier* is a Boolean value. It indicates if a data field is an identifier field of a substructure.

For each Communicative Event in the CED a message structure is required with information needed to express its behaviour.

C. Generating Test Cases from Communication Analysis

Our first motive for using Communication Analysis is to obtain a single model to specify the functionality of an IS and to generate the respective test cases. In this way the use of different artefacts by requirements analysts, testers and developers is avoided, thus making their work easier. As the events sequence describes the expected exchanges of messages between the actor and the system, this can be used to define the test cases. In particular, while the communicative events indicate the actions to be performed in a complete and uninterrupted way under certain constraints, the message structures for each communicative event contain references to the types involved that represent actors, or business concepts, the relationships between them and parameterized messages with data types existing in the conceptual schema of the system (the class diagram and state machines).

However, this forces the requirement analyst to be precise and rigorous in the semantics given to each CA concept and, thus, may not be so easy to build. To reduce this complexity, we use the existing editor tool [15], which is a Domain Specific Language to create a CED and introduce a message structure for each communicative event.

Our second motive comes from the fact that requirements-based testing [3], particularly model-driven testing [16], is being increasingly used. There is thus a need for a systematic approach to generating test cases from requirements model.

Our third motivate is in the context of MDD, where it is possible to obtain a test model from a requirements model by means of model transformations, so that the process can continue to generate the executables test cases. This means when a modification is made in the requirements model, not only is the test model automatically re-generated, but so are the concrete test cases.

Our approach is therefore designed to make generating test models and abstract test cases from requirement models easier.

IV. META-MODELS AND TRANSFORMATION RULES

This section describes the different meta-models and the transformation rules used in our proposal.

[1] It specifies the type of information that field contains (e.g. number, text).

[2] OCL: http://www.omg.org/spec/OCL/

A. Test Model Meta-model (TMM)

The meta-model defines the abstract syntax of the test model and the transformation rules are defined according to it. Fig. 3 shows the meta-model for generating the test models. A TMM instance is the PIT for our MDT proposal and consists of the principal class *TestModel*. This class has a *name* that identifies it, which is the same as the requirements model name. Some meta-model classes (i.e. TestModel, Precedence, TestComponent, TestItem and Parameter) inherit the attribute *name* from the class *Element*.

The class *TestModel*, which models the test component, consists of one or more classes of the type *TestComponent*, which contains all the necessary items to test the respective communicative event. It also contains the attribute *eventReference* which has one trace with the communicative event of the requirements model.

The class *Precedence* allows two test components (TC_i and TC_j) to be related, where TC_i (*source*) must necessarily occur before TC_j (*target*). The class *TestItem* models the test items; it contains *owner* (it is the object to which the test item belongs). A *TestItem* can be specialized in the type *Call* (i.e. services, triggers and links) and *Assertion*.

The class *Call* has the attribute *type* to indicate the type of service, trigger or link. The class *Assertion* has the attribute *constraint* modelled. The classes *Service* and *Trigger* may or not have *Input Parameter*. However, only class *Service* can have an *Output Parameter*. Additionally, the class *Service* has the attribute *visibility* (i.e. public, private) to define its behaviour.

The class *Parameter* models parameters that have the class *Call*, and contains the attributes *type*, *lowerBound* and *upperBound* (these are used as boundary values of the parameter range). A Parameter can be specialized as *Input* (optional in *Service* and *Trigger*) and *Output* (required for Services). Finally, the class *Link* allows testing the relationship between two objects specified by both the *Input Parameter* and *Output Parameter* values.

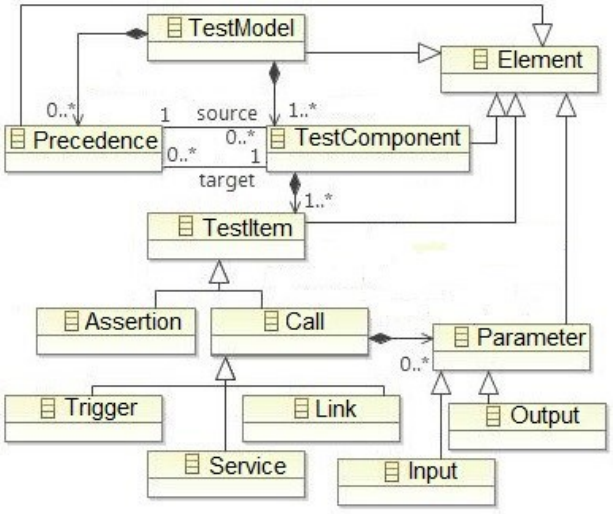

Fig. 3. Test model meta-model

B. Abstract Test Cases Model Meta-model (ATCMM)

For the second transformation from PIT to PST (instance of the Meta-model of the Abstract Test Cases Model) we have defined another meta-model (see Fig. 4) that allows platform-specific properties required in the abstract test cases to be added, such as resolving data type for the attribute *type* of the class *Parameter,* adding the attributes *type* in the class *TestCase* and *targetLanguage* in the class *TestModel*. Additionally, the class *Data* permits values for parameter as well as the expected values of the Test Items to be related. In class *TestItem* the attributes *verdict* (i.e. none, pass, inconclusive, fail and error) and *type* (i.e. positive and negative) has been added. This model is the PST for our MDT proposal.

C. Transformation Rules

A transformation definition is a set of transformation rules that together describe how models in the source language can be transformed into models in the target language [5].

Twelve transformation rules (R) were defined for the first transformation (PIM to PIT). Fig. 5 (Part a) shows the mappings between the RM to TM concepts. TABLE 2 shows an example of a transformation rule created in Atlas Transformation Language (ATL)[3], which transforms the primitives *EventVariant* from RM to the primitives *Test Component* of the TM. For the transformation (PIT to PST) the second meta-model is used (see Fig. 4). Eleven transformation rules (R') were defined to traverse the test model and generate test items of each test component grouped into the different abstract test cases (see Fig. 5 Part b).

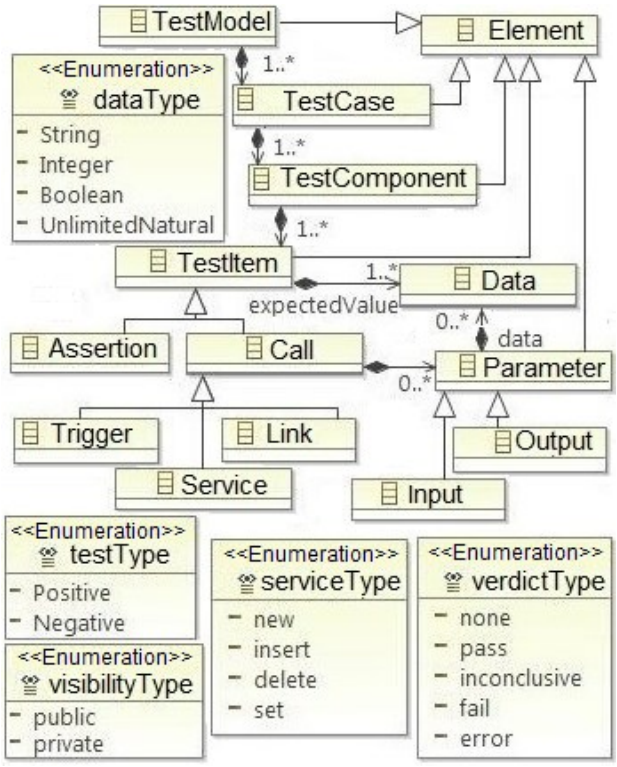

Fig. 4. Meta-model of the abstract test cases

[3] ATL: http://www.eclipse.org/atl/

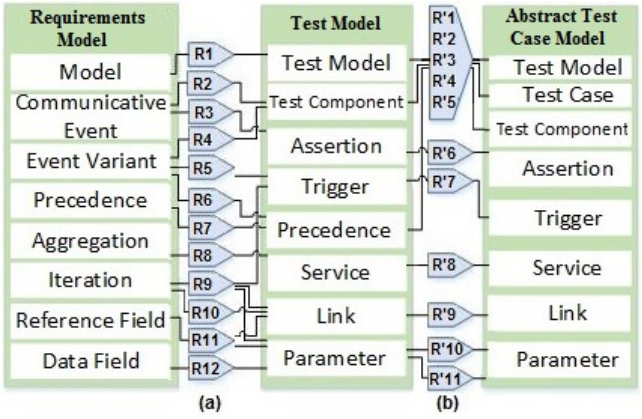

Fig. 5. Mappings for a) RM to TM primitives and b) TM to ATCM primitives

This transformation allows the test model to be refined (generate the abstract test cases) and enriched with test-specific properties, such as those mentioned in the meta-model description.

V. Overview of the Generation Process

A. *Step 1: Generation of Test Model*

As seen in Fig. 6, the derivation strategy starts with loading the requirement model (RM), the requirement meta-model (RMM) and test model meta-model (TMM). The XML Metadata Interchange (XMI)[4] file stores the RM which is created by the requirements engineer, based on the Communication Analysis method previously introduced in Section III. Once the models have been loaded, the requirement model is transformed into the test model (TM), where the test cases are ordered according to the precedence relationships.

Given the transformation rules defined in Section IV, the first step consists of processing each primitive of the CED (Fig. 2) together with the associated message structure (see the example in TABLE 1) in order to generate the Test Items grouped by each Test Component, which are related to each communicative event (all-events coverage). Each Test Component is made up of a set of statements (i.e. assertions as preconditions, call of services and triggers and call of links) to be tested. The different test components are then integrated in a single Test Model. Fig. 7 shows the Test Model generated for our example (OCR system) with the corresponding detailed test items only for the test component called SUBMIT_PAPER (TC_3).

TABLE 2. Transformation Rule 1

```
rule R4_CE_with_EventVariant2TestComponent{
from cametamodel:
cametamodel!CommunicativeEvent(self.has_EventVaria
nt(cametamodel))
to
testcomponent: distinct tcmetamodel!TestComponent
foreach (c in cametamodel.specialisations)(
name<- self.format_to_without_space(c.name),
eventReference<-cametamodel.name+':EventVariant',
testItems<-
self.R5_EventVariant2Trigger(cametamodel))}
```

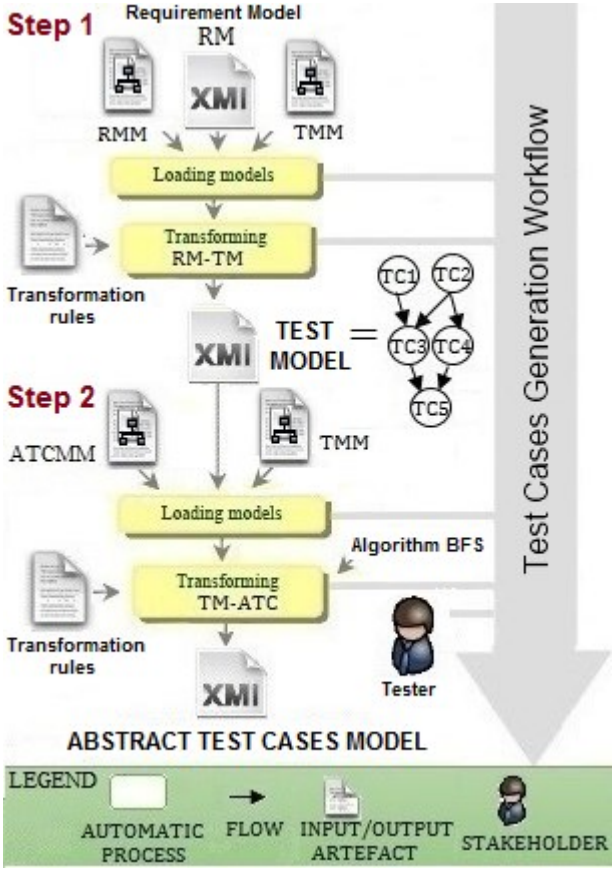

Fig. 6. Test cases code generation workflow

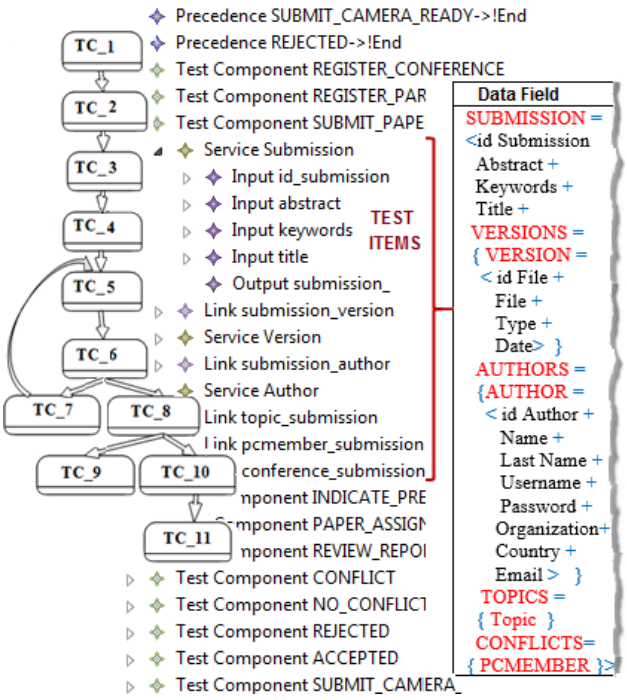

Fig. 7. TM with two detailed TCs for MEM1 and SUB1 respectively

[4] http://www.omg.org/spec/XMI/

B. Step 2: Generation of the Abstract Test Cases Model

The second step consists of processing the test model obtained to generate the abstract test cases. For this purpose, the models and meta-models (i.e. TM, TMM and ATCMM) have to be loaded.

Our approach traverses the test model using an algorithm based on Breadth-First Search (BFS) [17]. This algorithm was adapted to generate the different test item sequences from the test model. The different sequences are generated considering the alternative paths (i.e. event variant) in the model.

This process then generates 3 abstract test cases (Abs_TCase), each one with 9, 8 and 7 test components, respectively, and each test component with 27, 24 and 23 test items, respectively (see Fig. 8).

These two steps permit the test model and abstract test cases to be generated from a requirements model.

The requirements covered by the generated abstract test cases are:

- Contact requirements; assertions (preconditions).
- Communication requirements; triggers and services with parameters and their data types.
- Reaction requirements: object links.

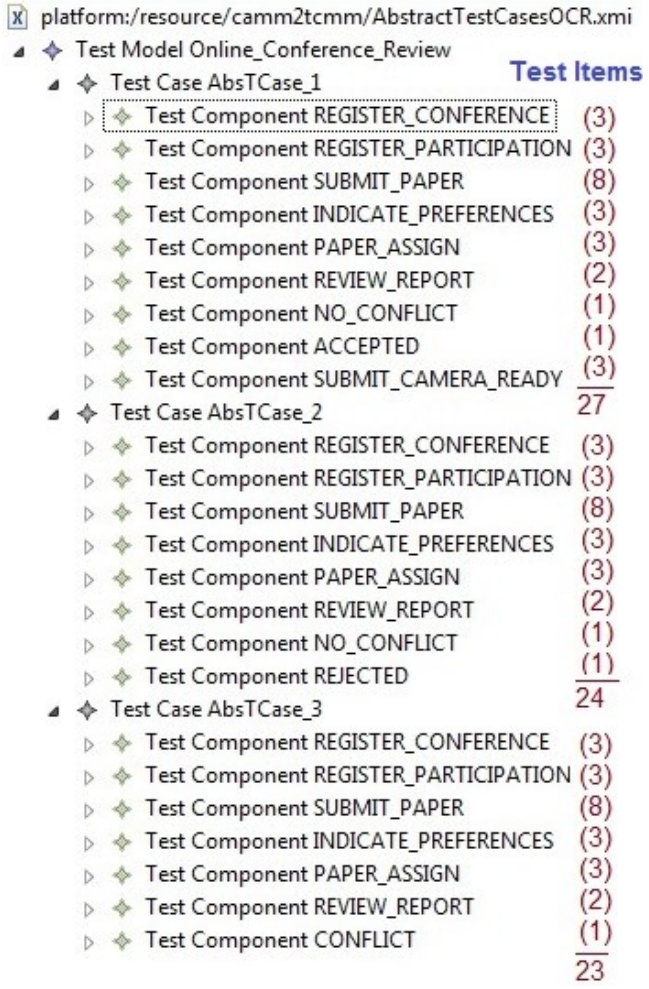

Fig. 8. Abstract Test Cases for OCR system

VI. Related work

In this paper we define an MDT approach for automatically generating a set of abstract test cases from a requirements model, which will be used to validate the requirements in conceptual schemas in an MDD environment.

Other approaches to generating test cases from functional requirements have been developed, such as those summarized by Escalona et al. [18]. However, only a few approaches can automatically generate test cases from requirements [19].

The major differences between our approach and the others are: that we focus on automated test model generation using MDT, while the others automate the tracing tasks for manual testing (e.g. tracing out a specific scenario and the tasks of inputting data and expected result preparation for each scenario). Also, our approach applies the test cases on CS and the other approaches on system code.

There are also works which focus on CS testing [20] [21], [22], [23], however only one of these [20] validates the CSs with respect to requirements, and none of them is integrated into an overall MDT process, unlike our proposal.

On the other hand, although there are many model-based testing approaches, such as those summarized by Utting et al. [4], we only consider model-based testing that follows an MDE paradigm (MDT), in which the test cases are derived from models and not from the system code [4], [24], [16].

However, the major differences with our proposal are: (i) the level of abstraction for testing artefacts (code level), in our case the testing is for CSs; (ii) testing purpose (verify the correctness of the system), in our case validating the correctness, completeness and consistency of CS with respect to requirements; (iii) these works use design models that are parts of CS (e.g. class diagram, sequence diagram, component diagram) to automatically derive the test cases, in our work the CSs are the artefacts (CSUT) for testing; (iv) we take advantage of a requirements model, allowing us to specify IS as well as the test cases in an MDE context, hence facilitating the requirements validation phase.

VII. Conclusions and Future Work

This paper presents an approach for automatically generating abstract test cases from a requirements model. This means testers and developers can be use the same artefacts and reduces the costs of the testing process.

A Model transformation strategy (MDT) has been defined to derive initial versions of test models and abstract test cases from Communication Analysis requirements models. To do this, twelve transformation rules were defined to facilitate the generation of the test models; and eleven refinement rules were defined for obtaining the abstract test cases from the test model.

An overview of our approach has been illustrated with a case example of an online Conference Review System, in which 3 abstract test cases are obtained from the test model. The abstract test cases are formed by 9, 8, and 7 test components respectively, each one with 27, 24 and 23 test items (i.e. assertions, services, trigger and links). The test cases cover the requirements at the communicative iteration level (i.e. contact, message and reaction) related to the communicative events.

With the purpose of obtaining a "good" set of abstract test cases, we plan to conduct various experimental studies to validate the completeness (e.g. [25]), correctness and scalability of our proposal.

Additionally we will try to assess the cost impact of the testing process using our approach.

Finally, we will test a number of strategies to concretize our abstract test cases. Executable test cases will also be obtained using the Alf model execution language [8].

ACKNOWLEDGMENTS

This work has been supported by The Secretary of Higher Education, Science and Technology (SENESCYT: Secretaría Nacional de Educación Superior, Ciencia y Tecnología) of the Republic of Ecuador.

REFERENCES

[1] K. Beck, Test-Driven Development by Example, Pearson Education, 2003.

[2] M. Wynne and A. Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Programmers, 2012.

[3] P. Skoković and M. Rakić-Skoković, "Requirements-Based Testing Process in Practice," IJIEM, vol. 1, no. 4, pp. 155-161, 2010.

[4] M. Utting, A. Pretschner and B. Legeard, "A taxonomy of model-based testing approaches," Softw. Test. Verif. Reliab, pp. 1-15, 2010.

[5] Z. Dai, "Model-driven testing with UML 2.0," in Computer Science at Kent, 2004.

[6] ISTQB, "Standard glossary of terms used in software testing," 2012.

[7] S. España, "Methodological integration of Communication Analysis into a Model-Driven," Valencia, 2011.

[8] A. Olivé, Conceptual Modeling of Information System., Springer, 2007.

[9] OMG, "Semantics of a Foundational Subset for Executable UML Models (FUML)," OMG, 2011.

[10] OMG, "Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF)," OMG, 2013.

[11] S. España, A. González and O. Pastor, "Communication Analysis: a Requirements Engineering Method for Information Systems," in *CAiSE*, Amsterdam, The Netherlands, 2009.

[12] O. Pastor and J. Molina, Model-Driven Architecture in practice: a software production, Springer,, 2007.

[13] S. España, M. Ruiz and A. González, "Systematic derivation of conceptual models from requirements models: a controlled experiment," in *in Sixth International Conference on Research Challenges in Information Science (RCIS)*, 2012.

[14] A. González, M. Ruiz, S. España and O. Pastor, "Message Structures: a modelling technique for information systems analysis and design," in *WER*, 2011.

[15] M. Ruiz, S. España, A. Gonzalez and O. Pastor, "Análisis de Comunicaciones como un enfoque de requisitos para el desarrollo dirigido por modelos," in *DSDM*, Valencia, España, 2010.

[16] M. Mussa, S. Ouchani, W. A. Sammane and A. Hamou-Lhadj, "A Survey of Model-Driven Testing Techniques," in *QSIC*, Jeju, Korea, , 2009.

[17] J. Kleinberg and E. Tardos, Algorithm Design, Boston: Pearson Education, 2006.

[18] M. J. Escalona, J. J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres and F. J. Domínguez, "An overview on test generation from functional requirements," *Journal of Systems and Software,* vol. 84, no. 8, p. 1379–1393, 2011.

[19] C. Nebut, F. Fleurey, Y. Le Traon and J.-M. Jezequel, "Automatic test generation: a use case driven approach," in *Software Engineering, IEEE Transactions on* , 2006.

[20] A. Tort and A. Olivé, "An approach to testing conceptual schemas," *Data & Knowledge Engineering,* pp. 598-618, 2010.

[21] D. A. Sadilek and S. Weißleder, "Testing Metamodels," in *Model Driven Architecture – Foundations and Applications*, vol. 5095, Springer Berlin Heidelberg, 2008, pp. 294-309.

[22] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh and R. France, "Testing UML designs," vol. 49, pp. 892-912, 2006.

[23] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France and A. Andrews, "A tool-supported approach to testing UML design models," in *Proc. of the ICECCS*, CO, USA, 2005.

[24] B. P. Lamancha, M. Polo, D. Caivano, M. Piattini and G. Visaggio, "A Model Based Testing Approach for Model-Driven Development and Software Product Lines," vol. 55, pp. 301-319, 2013.

[25] M. F. Granda, "An experiment design for validating a Test Cases Generation Strategy from a Requirements Model," in *Empire*, Karlskrona, Sweden, 2014.

[26] R. Van de Stadt, "CyberChair," [Online]. Available: http://www.borbala.com/cyberchair/. [Accessed 03 2014].

[27] P. D. V. N. T. Mohagheghi, "Definitions and approaches to model quality in model-based software development– A review of literature," vol. 51, pp. 1646-1669, 2009.

APPENDIX I. DESCRIPTION OF ONLINE CONFERENCE REVIEW SYSTEM

In order to exemplify the application of our generation approach of a test model throughout this paper, we will use a system called Online Conference Review (OCR) based on the description of the CyberChair System [26].

In this system there is a program committee chair (PcChair) that determines the topics of interest and selects the members of the program committee (PcMember).

Authors are responsible for sending their paper submissions indicating potential conflicts of interests with PcMembers.

The PcChair allocates items to the members of the program committee (PcMember), resolving the conflicts of interest, if applicable.

Reviewers (PcMember) are responsible for assessing the submissions assigned to them and recording their assessments.

PcChair resolves the conflict of evaluations, if applicable, and sends notifications of acceptance or rejection of the submission authors.

Finally, if the submission is accepted the authors of the papers will be invited to revise and submit (camera-ready) improved versions of their papers.