Derivation of test objectives automatically

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

Department of Computer Languages and Systems.

University of Seville.

{javierj, escalona, risoto, jtorres}@lsi.us.es

A vital task of software development is to test the correct implementation of functional requirements. Use cases are widely used artefacts that define the functionality of a software system in early stages of the development process. This paper exposes the lack of automatism in existing approaches that deal with the derivation of test cases, and introduces a new approach and tool to derive systematically test objectives from the use cases of the system under test.

KEYWORDS

System testing, test objectives, generation of test cases, open-source tools.

1. Introduction

The growing complexity of software systems increases the need to assure their quality. The system testing is a technique that helps to ensure the quality of software systems. It is defined as a black-box procedure to verify the satisfaction of the requirements of the system under test (SUT) [3]. Several kinds of tests might be performed during the system testing phase. Some of them are: navigational testing, reliability testing, usability testing, etc [3]. This paper is focused on the functional testing from the point of view of external actors and, specifically, from a human user point of view through a graphical interface. Thus, a test case substitutes an actor of the system and simulates the interactions with the actor to check that this system does what it is expected to do. For this reason, the main artefact to

obtain system test cases is its functional requirements since they describe all the expected behaviours that have to be tested by the system test cases.

Functional requirements are often defined as use cases. Use cases offer a general vision of the system. They are easier to study and validate for non-technical users. In early development phases, when requirements are being discovered, defined and negotiated, it is quite easier to modify use cases defined as prose or structured natural language, than to make changes in formal requirements.

Most of software testing in industry is conducted at the system level. However, the most formal research has been focused on the unit level [15]. Thus, most system-level techniques are only informally described. The system testing requires a formal process to identify the most important test cases and to measure the grade of efficiency. Another important problem is that the system testing is performed at the end of the development process, when the system is codified. Due to a tight schedule time, the design and execution of system testing are frequently only superficially performed or not at all. This paper tries to resolve one of these lacks. It proposes a systematic process to derive test objectives from use cases, in order to verify the successful implementation of these use cases in the final software system.

A test objective is a named element that describes what should be tested [21]. The test objectives derived from use cases define what we have to test to ensure the right implementation of the use case and the complete implementation of the use cases. Designing good objectives is a vital task for the testing as well as for the software development. However, the UML Testing Profile [21] does not define any notation to represent test objectives. We have resolved this gap using activity diagrams and sequences of activities.

An activity diagram has several advantages over other UML diagrams. UML sequence diagrams need information about the components of the systems and the signatures and parameters of the calls among the components. Moreover, sequence diagrams do not allow to represent alternative or erroneous scenarios. UML state diagrams are focused on the states and transitions of the system, but they do not clearly show the interactions between the actors and the system.

This paper is organized as followed: section 2 briefly describes several surveys about approaches that deal with the derivation of test objectives. Then, section 3 proposes a model and template to define use cases. Section 4 introduces a real application using a case study. Then, section 5 describes the process to derive test objectives from use cases using the template of section 3, and illustrates the process with the system described in section 4. Section 6 describes other related approaches. Finally, section 7 lists the conclusions and future works.

2. State of the art

While writing this paper, we have identified several approaches to generate test objectives. Two surveys that analyse and compare 21 different approaches (in to-

tal) may be found in [5] and [10]. Next paragraphs reference some of these approaches. A complete list of references may be found in both surveys.

The existing approaches might be divided into three groups depending on the artefacts used for the generation of test cases. The first group includes approaches that generate test objectives directly from use cases, like [2] and [11]. The second group includes approaches that generate a behavioural model from the use cases and derive test objectives from them, like [18], [13] or [17]. Some of the notations used in this group are: activity diagrams, state-machines diagrams, use case transitions systems or scenario trees. The third group describes approaches focused on variables and test values. These approaches identify variables in use cases and perform partition of the domains, like [2] and [16].

The approaches might also be divided into two groups depending on the scope of the generated test objectives. In the first group, we find approaches that generate test objectives from isolated use cases, like [2], [11] or [17]. The second group includes approaches that generate test objectives from sequences of use cases, like [18] or [13]. However, none of the approaches of the first group might automatically derive behavioural models and test objectives from use cases. We can also point out a lack of supporting tools. In the second group, the reference [13] generates test objectives in an automatic way for sequences of use cases, not for isolated use cases. These facts justified a new approach to generate a behavioural model and to derive test objectives in an automatic way. Next section describes the model and template used to define use cases.

3. A use case model and template for testing.

The systematic generation of test objectives implies some drawbacks. One of them is the need for defining a concrete model for the use cases that may be manipulated in a systematic and automatic way, without loosing the advantages of using prose text. A widely used solution is to structure the use cases in templates that combine prose texts with a concrete structure and fields.

We use the requirement model proposed by the Navigational Development Technique (NDT) [6]. Although NDT is focused on the navigational aspect of web and hypermedia systems, it offers a complete, formal and flexible requirement model. This model may be used with all kind of information systems, as demonstrated [8]. The requirement model of NDT also proposes a template to define functional requirements like use cases. This template is quite similar to the ones proposed by other authors. We have chosen the NDT requirement model and template for three main reasons. First, this model is based on a formal UML metamodel [6], [14]. Second, the templates proposed in NDT only contain the more relevant elements for the use case definition and test objectives derivation, and might be easily extended. Finally, the NDT requirement model has been applied in many real and complex projects like [8]. NDT has also a supporting tool called NDT-Tool.

We have performed a minimal extension to the NDT templates to improve it testability and to allow a systematic and automatic process to derive test objectives. An example of the NDT extended template model is shown in table 1. A real use case defined by this template may be found in table 2.

Table 1. Use case template.

Name	UC-01. …
Precondition	…
Main sequence	1. The actor….. 2. The system … …
Errors / alternatives	1.1.i. If the system … then … and the result is …. 2.1.p. If the actor … then … and the result is …. 3.1.i. At any time, the [system/actor] may …. then …. and the result is …
Results	1. System … …
Post condition	…
Reliability	…
Priority	…

Several fields in the template, like precondition, post-condition, etc., are common to other templates approaches and are widely known and described in other papers and books [4]. Next paragraphs describe special fields and particular characteristics of the extension. The ideas exposed in next paragraphs may be easily adapted to other requirement techniques and models.

Name: Te name describes the goal of the use case. Every use case has a unique identifier that starts with "UC" letters following one number.

Main sequence: The main sequence is composed of the steps of the use cases execution to allow the actor to obtain his objective. Every step of the main sequence is composed of an identifier (the steps are numerated consecutively starting from number one), who performs the step (an actor or the system) and the performed action.

Errors / alternatives: These steps describe the behaviour of the system when an error is found or some alternative flow may be executed. An error or alternative step has got the same elements as the main sequence and some additional elements.

The identification of an error or an alternative step is composed of two numbers. The first number must be an existing step of the main sequence. The second number allows to distinguish among different alternatives or errors of the same step. Every step in an error / alternative section must have an evaluated condition to decide if the error or alternative step might be executed, and also to decide the executed action when the condition is true, and the result, for example the end of the use case or the repetition of a set of steps. Sometimes the action might be the same as the result and the result might be omitted.

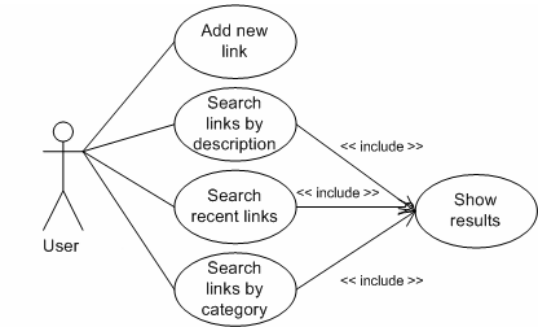

Fig. 1. A use case model.

The conditions of an error or alternative are classified in preconditions or invariants. A precondition is evaluated before the step starts its execution. An invariant is evaluated during the execution of the step.

Result: It indicates which steps (in main sequence and in alternative or erroneous sequences) end with the use case and which is the obtained result to the main actor. Some use cases have not a visible result, or their definition is out of the scope of the use case. This fact is pointed out showing that the result is not a visible result.

Next section describes the system used to apply the approach presented in this paper and it exposes some examples of use cases defined with the template of table 1.

4. Case study

The system under test is a web application that allows to manage an on-line link catalogue (found in www.codecharge.com). The system includes two actors: the user and the administrator. However, in this case study, we will only consider the user actor. The UML Use Case diagram of the user is shown in figure 1.

The use cases of the case study are the following ones: "Search link by description" and "Show results". Due to their inclusion relation, both use cases are defined using one instance of the template of section 3 (table 2).

Table 2. The use case: "Search link by description".

Name	UC-02. Search link by description
Precondition	No.
Main sequence	**1.** The user asks the system for searching links by description. **2.** The system asks for the description. **3.** The user introduces the description. **4.** The system searches for the links which matches up with the description introduced by the user. **5.** The system shows the results.

Errors / alternatives	**3.1.i.** At any time, the user may cancel the search, then the use case ends. **4.1.p.** If the actor introduces an empty description, then the system searches for all stored links and the result is to continue the execution of this use case. **4.2.i.** If the system finds any error performing the search, then an error message is shown and this use case ends. **5.1.i.** If the result is empty, then the system shows a message and this use case ends.
Results	**5.** The system shows the results of UC-05. **3.1.i.** Out of the limits of this use case. **4.2.i.** Error message. **5.1.p.** Message of no found results.
Post condition	No

The alternative steps are annotated with 'p' if they are preconditions and 'i' if they are invariants. Next section describes how to obtain test objectives from use cases and how to apply the process over this use case.

5. Test objectives from use cases

To derive test objectives, first, a behavioural model from a use case is built. Then, the behavioural model is rounded trip to identify the test objectives. Point 5.1 describes the generation of the behavioural model. Then, point 5.2 describes how to derive test objectives, and point 5.3 describes how to manage the coverage of the use case by the test objectives.

5.1. Building of a behavioural model

The first task is to build a behavioural model from the use case. As mentioned in section 1, a behavioural model is a UML activity diagram. This model represents the different scenarios or instances of a use case. Next paragraphs describe the steps to build a behavioural model from the use case which is defined with the template of table 1.

In the main sequences, each step is an activity of the behavioural model. A transition is added through two consecutive steps. A behavioural model has got, at least, one ending point. Figure 2 shows an example of a behavioural model from the main sequences of the use case of table 2.

Each alternative or error step is a decision node. If the alternative or error step is a precondition, it is added before the related action. If the step is invariant, it is added after the related action. If the alternative or error step performs an action, the latest will be a new activity. If the result of the use case is to repeat a previous step, a transition to the activity representing the step is added. The condition evaluated in the alternative is attached to a decision node. Alternative and errone-

ous steps are also classified into three categories. First one, called end, indicates that the alternative ends the use case. An example is shown in figure 3 (alternative 1). Second one, called goto, indicates that the alternative repeat a previous activity. Third one, called action, indicates that he alternative performs a new activity. An example is shown in figure 3 (alternatives 2, 3). Every category is processed in a different way, as can be seen in algorithm "BuildBehaviourModel".

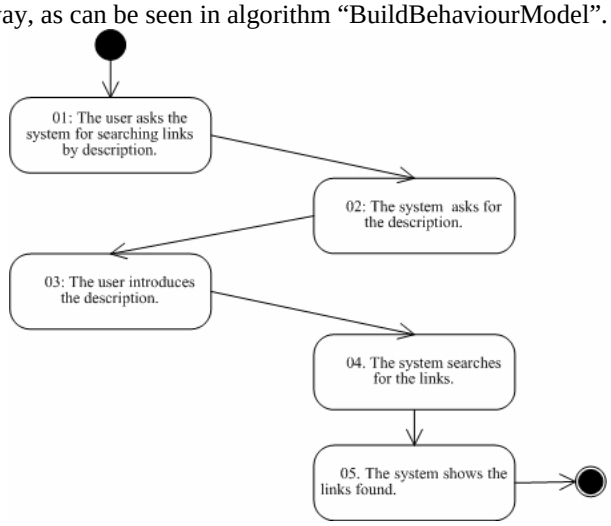

Fig. 2. Activity model from the main sequence of use case.

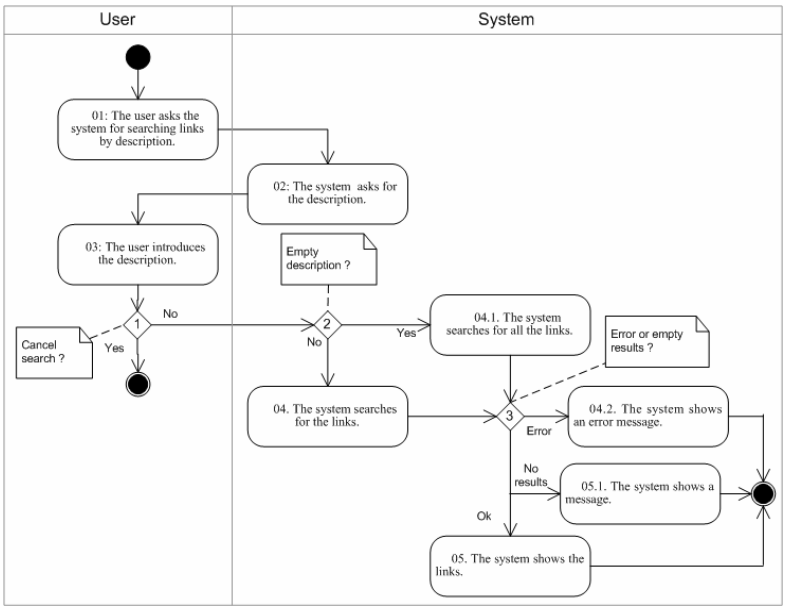

Fig. 3. The complete behavioural model.

If there is a sequence of decisions nodes, all of them belong to the same actor (including system). They should be merged into one decision node. Finally, the activities are classified by classifiers. The behavioural model will have a classifier for each actor and one more for the system. Each classifier will contain the activities performed by the actor or by the system. Figure 3 shows the final result.

Algorithm "BuildBehaviourModel" describes the algorithm used in the supporting tool to generate the test objectives. A model variable is an activity graph as defined in UML [20], whereas a step variable is a step from a use case, as defined in section 4. Helper functions have a self descriptive name and their definition has not been included.

```
algorithm BuildBehaviourModel
var     model : ACTIVITYGRAPH
        alternativeSteps : LIST[USECASESTEP]
        step : USECASESTEP
init
   foreach (step in useMase.mainSequence)
        alternativeSteps = useCase.getAlternatives(pre, step)
        if ( not_empty(alternativeSteps) )
                traverse_alternativeSteps(behaviourModel, alternativeSteps)
        end if
        behaviourModel.addActivity(step)
        alternativeSteps = useCase.getAlternatives(inv, step)
        if ( not_empty(alternativeSteps) )
                traverse_alternativeSteps(behaviourModel, alternativeSteps)
        end if
   end foreach
end init
function traverse_alternativeSteps(behaviourModel, alternativeSteps)
        alternative : STEP
        decision = behaviourModel.addDesicion(alternativeSteps))
        foreach ( alternative in alsternativeSteps)
                if ( is_activity(alternative.action) )
                        node = behaviourModel.addActivity(alternative.action)
                end if
                if ( is_end(alternative.action) )
                        node = behaviourModel.addActivity(activityEnd)
                end if
                if ( is_gotoActivity(alternative.action) )
                        node = behaviourModel.getActivity(alternative.action)
                end if
                behaviouralModel.addTransition(decision, node)
        end foreach
end function
```

If each step of the use case defines only one activity, then the maximum number of nodes (activities and decisions; start and end nodes are not included) of a test objective model is: the number of steps in main sequences plus (number of alternative and error steps x 2). From the use case of table 2, we can see that the maximum number of nodes is 5 + (4 x 2) = 13 nodes. The behavioural model of figure 3 shows only 11 nodes, because step 3.1.i does not generate any activity and steps 4.2.i and 5.1.p have been combined in the same decision (decision 3 in figure 3).Next point describes how to identify loops in the behavioural model.

5.2. Derivation of test objectives

After the building of a behavioural model, the test objectives are systematically derived from them. The test objectives are defined as paths over the behavioural model. These paths might also be expressed like activity diagrams or text. An example of test objectives is shown in table 3 and figure 4.

However, a test objective is not a test case because it cannot be executed over the system under test. The test objectives have to be completed with test values and expected results, and should be executed in a test director tool or translate into test scripts. An example of how to implement test objectives may be found in [13] and [9].

Several coverage criteria might be chosen to generate sequences from a graph. For example, two classic criteria are all-nodes and all-edges criteria. However, the coverage criterion selected for this approach is the all scenarios criterion (AS). A set of test objectives satisfies the whole scenarios criterion for a behavioural model if each scenario involved in the use case is exercised by one and only one test objective. A scenario is an instance, or a concrete execution, of a use case.

The AS coverage criterion assure that all the obtained objectives are reachable and none of the objectives is repeated. Two test objectives are the same when they have the same number of activities appearing in the same order.

Algorithm “BuildTestObjectives” describes the algorithm used in the supporting tool to generate test objectives. Helper functions have a self descriptive name and their definition has not been included.

```
algorithm BuildTestObjectives
var     objective : PATH
        objectives : LIST(PATH)
init
        objective = < empty >
        objectives = < empty >
        traverse(initialNode, path)
end init

function traverse(in node, inout objective)
        if ( is_desicion(node) )
                traverse_desicion(node, objective)
```

```
                exit function
        end if
        objective.add(node)
        if ( isEnd(node) )
                objectives.add(objective)
                exit function
        end if
        nextNode = next_node(node)
        traverse(nextNode, objective)
end function

function traverse_desicion(in node, inout objective)
        foreach (alternative in node.alternatives)
                path.add(alternative)
                nextNode = next_node(alternative)
                traverse(nextNode, objective)
        end foreach
end function
```

A set of test objectives are obtained after the application of algorithm 2 on the behavioural model. The test objectives with a AS coverage are listed in the following table 3. The id 1 path is the test objective of the main sequence of the use case.

Table 3. Derived test objectives.

Id	Path
1	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(No error & Results) -> 05
2	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(No error & No Results) -> 05.1
3	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(Error) -> 04.2
4	01 -> 02 -> 03 -> D1(No) -> D2(Yes)-> 04.1 -> D3(No error & Results) -> 05
5	01 -> 02 -> 03 -> D1(No) -> D2(Yes)-> 04.1 -> D3(No error & No Results) -> 05.1
6	01 -> 02 -> 03 -> D1(No) -> D2(Yes)-> 04.1 -> D3(Error) -> 04.2
7	01 -> 02 -> 03 -> D1(Yes)

The test objectives might also be represented by activity graphs. Figure 4 shows the activity diagrams that match paths 1 (4(a)) and 7 (4(b)).

As mentioned below, the objectives of table 7 cover 100% of scenarios of the use case. The next section describes a criterion to select the desired coverage.

5.3. Coverage of use cases.

The coverage of test objectives measures the number of scenarios from a use case with an attached test objective. The coverage of test objectives might be calculated with the formula of figure 5.

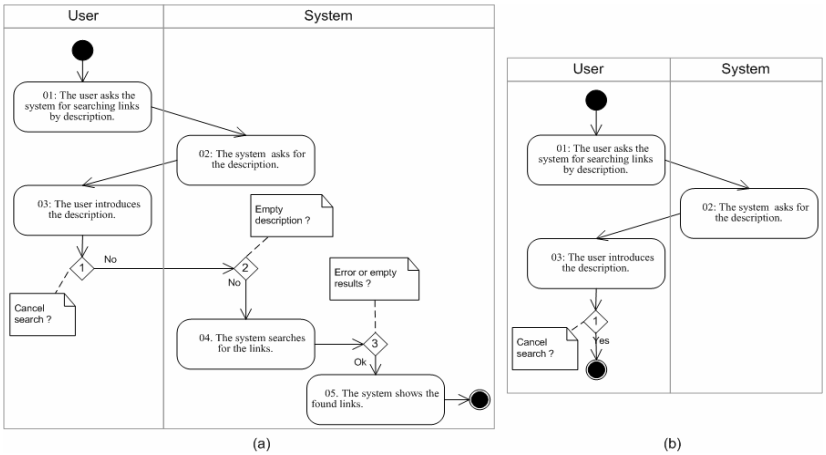

Fig. 4. Test objectives as activity graphs.

A higher coverage implies more test objectives and more test cases. A coverage of 1, means that every possible scenario has one test objective and will have, at least, one test case (as seen in the section below).

$$\frac{\text{Number of test objectives}}{\text{Number of scenarios}} = \text{Test coverage}$$

Fig. 5. Measure of the test objective coverage.

The coverage, and the number of test objectives, might be determined by the relevance or frequency of the use case. Both elements are included in the template model proposed in section 3. A coverage criterion is shown in table 4.

Table 4. Coverage criterion.

Priority	Coverage
0	No test objectives are generated from the use case.
1	Only one test objective is generated from the main sequences.
2	Main sequence and all decisions nodes in actor classifiers.
3	All the test objectives are generated.

Next section exposes conclusions and ongoing works.

7. Conclusions

Test objectives are basic for a successful testing. They indicate which test cases have to be built to test the implementation of a use case. This paper has presented

a new approach of the systematic derivation of test objectives for use cases. In section 2, we justified the need for a new approach because we have not found any process to derive automatically test objectives from isolated use cases.

A previous work described how to generate test cases from use cases for web application using existing approaches [9]. However, this approach is highly based on manual work and the decisions of test engineers. The automatic generation of test objectives presented in this paper continues this research and is also the first step to obtain a complete process for the generation of test cases, as we will see in ongoing works. Our approach generates the same test objectives as the referenced approaches of section 2. However, the main advantages are the definition of a semiformal model to define use cases and the automatic generation of test objectives. All derived objectives are also reachable, as seen in section 4.3. The approach does not generate repeated objectives. The algorithms described in section 4 have been implemented in a prototype tool. This tool may be downloaded from www.lsi.us.es/~javierj/ and is being improved with loop management and XMI support.

Future works aim at extending the presented approach. Our final goal is to generate test scripts from test objectives in an automatic way. Some preliminary works about the generation of test scripts may be found in [9].

REFERENCES

[1] Bertolino, A., Gnesi, S. 2004. PLUTO: A Test Methodology for Product Families. Lecture Notes in Computer Science. Springer-Verlag Heidelberg. 3014 / 2004. pp 181-197.

[2] Binder, R.V. 1999. Testing Object-Oriented Systems. Addison Wesley.

[3] Burnstein, I. 2003. Practical software Testing. Springer Professional Computing. USA.

[4] Cockburn, A. 2000. Writing Effective Use Cases. Addison-Wesley 1st edition. USA.

[5] Denger, C. Medina M. 2003. Test Case Derived from Requirement Specifications. Fraunhofer IESE Report.

[6] Escalona M.J. 2004. Models and Techniques for the Specification and Analysis of Navigation in Software Systems. Ph. European Thesis. Department of Computer Language and Systems. University of Seville. Seville, Spain.

[7] Escalona M.J. Martín-Pradas A., De Juan L.F, Villadiego D., Gutiérrez J.J. 2005 El Sistema de Información de Autoridades del Patrimonio Histórico Andaluz Proceedings of V Jornadas de Bibliotecas Digitales (JBiDi 2005). ISBN: 84-9732-453-6 Granada, Spain. September, 2005.

[8] Gutierrez J.J. Escalona M.J. Mejías M. Torres J. 2004. Aplicando técnicas de testing en sistemas para la difusión Patrimonial. V Congreso Nacional de Turismo y Tecnologías de la Información y las comunicaciones (TURITEC'2004). pp. 237-252. Málaga, Spain.

[9] Gutiérrez J.J., Escalona M.J., Mejías M., Torres J. 2005. A practical approach of Web System Testing. Advances in Information Systems Development: Bridging the gap between Academia and Industry. pp. 659-680. Ed. Springer Verlag Karlstad, Sweden.
[10] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. 2006. Generation of test cases from functional requirements. A survey. 4º Workshop on System Testing and Validation. Potsdam. Germany.
[11] Heumann , J. 2002. Generating Test Cases from Use Cases. Journal of Software Testing Professionals.
[12] Myers G. 2004. The art of software testing. Second edition. Addison-Wesley. USA.
[13] Nebut C. Fleury F. Le Traon Y. Jézéquel J. M. 2006. Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering Vol. 32. 3. March.
[14] Koch N. Zhang G. Escalona M. J. 2006. Model Transformations from Requirements to Web System Design. Webist 06. Portugal.
[15] Offutt, J. et-al. 2003. Generating Test Data from Sate-based Specifications. Software Testing, Verification and Reliability. 13, 25-53. USA.
[16] 16 T. J., Balcer M. J. 1988. Category-Partition Method. Communications of the ACM. 676-686.
[17] Ruder A. 2004. UML-based Test Generation and Execution. Rückblick Meeting. Berlin.
[18] Labiche Y., Briand, L.C. 2002. A UML-Based Approach to System Testing, Journal of Software and Systems Modelling (SoSyM) Vol. 1 No.1 pp. 10-42.
[19] Several authors. 2004. SWEBOK. Guide to the Software Engineering Body of Knowledge. IEEE Computer Society. [21] Object Management Group. 2002. The UML 2.0 Testing Profile. www.omg.org
[20] Object Management Group. 2003. Unified Modelling Language 2.0. www.omg.org
[21] Object Management Group. 2003. The UML Testing Profile. www.omg.org