Effectiveness Assessment of an Early Testing Technique using Model-Level Mutants

M. F. Granda
Computer Science Department, University of Cuenca
Ecuador
fernanda.granda@ucuenca.edu.ec

N. Condori-Fernández
Computer Science Department, VU University
The Netherlands
n.condori-fernandez@vu.nl

T. E. J. Vos
PROS, Universitat Politècnica de València
Spain
tvos@pros.upv.es

O. Pastor
PROS, Universitat Politècnica de València
Spain
opastor@pros.upv.es

ABSTRACT

While modern software development technologies enhance the capabilities of model-based/driven development, they introduce challenges for testers such as how to perform early testing at model level to ensure the quality of the model. In this context, we have developed an early testing technique supported by the CoSTest tool to validate requirements at model level. In this paper we describe an empirical evaluation of CoSTest with respect to its effectiveness in terms of its fault detection and test suite adequacy. This evaluation is carried out by model-level mutation testing using first order mutants (created by injection of a single fault) and high order mutants (containing more than one fault) with seven conceptual schemas (of different sizes) that represent the functionality of different software systems in different domains. Our findings show that the test suites generated by CoSTest are effective at killing a large number of mutants. However, there are also some fault types (e.g. WCO1, WCO3) that our test suites were not able to detect. CoSTest's effectiveness is affected by the mutant type that is executed; high order mutant types were more effective in terms of detecting fault types and test suite adequacy.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging** • **Software and its engineering → Empirical software validation**

KEYWORDS

Test Suite Effectiveness, Effectiveness Assessment, Mutation Testing, Conceptual Schemas Testing, Class Diagram Mutation

ACM Reference format:

M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor. 2017. SIG Proceedings Paper in word Format. In *Proceedings of ACM Woodstock conference, Karlskrona, Sweden, June 2017 (EASE'2017)*, 10 pages.
DOI: 10.1145/123 4

1 INTRODUCTION

Constructing software automatically from models or Conceptual Schemas (CS) is one of the current challenges in software engineering, especially in a Model-driven Engineering context [26]. A well-formed model, being an accurate representation of all the requirements for a system under construction, is a key factor in the successful development and production of the system. The development of a CS is an iterative process involving evaluation of the model, its accuracy and its improvement from the evaluation results.

Testing is a well-established technique that helps to accomplish this task and provides a level of confidence in the end product based on the coverage of the requirements achieved by the tests.

In this context, we defined an early testing technique for validating Conceptual Schemas in a Model-driven environment [14][13]. This technique covers: 1) test suite generation, 2) CS under test generation, 3) test execution and report generation with the faults detected and the coverage analysis. Therefore, the technique's effectiveness and adequacy of the test suite require to be evaluated.

Effectiveness in detecting faults can be evaluated by the types and number of faults that can be detected by the technique [28]. For assessing the adequacy of a test suite, mutation testing is a method that injects artificial faults or changes into a software product (mutant) and checks whether a test suite is "good enough" to detect these artificial faults. The adequacy level of the test suite can be measured by a mutation score that is computed in terms of the number of mutants killed (detected) by the test suite [18]. Killing a mutant means that the execution is stopped because a fault was detected or because it reaches an inconsistent state and cannot continue execution. Mutants are produced by using mutation operators that describe syntactic changes to the original software product. Mutants can be classified into two types: First Order Mutants (FOM) and Higher Order Mutants (HOM) [19]. Traditional mutation testing considers FOM created by injection of a single fault. HOM contain more than one fault. Jia and Harman claim that some HOMs are harder to kill than the FOMs

[18], and so we were interested in evaluating the effectiveness of CoSTest test cases in both mutant types.

Mutation testing was originally introduced by DeMillo et al. [5] and Hamlet [17], as a support technique for developing tests for software systems represented at the code level. However, it has also been applied to models at the design level, for example to Finite State Machines [9], State Charts [11], Activity Diagrams [10], and Network protocols [20].However, there is no empirical evidence on the effectiveness of mutation testing in improving test suites for Conceptual Schemas.

This paper uses a mutation-testing based approach to evaluate the fault detection effectiveness of an automatically generated test suite to test a given CS. This means the CS is mutated and not the code!

In a previous paper [12],we proposed a set of 50 mutation operators specifically designed to generate mutants for UML Class Diagram-based CS and we evaluated the usefulness of an effective subset of mutant types of 18 mutation operators to inject defects into a CS. For this, we developed 1) the MµtUML tool (Mutation for UML) [16] for the generation and parsing (i.e. syntax analysis) of first order mutants (i.e. mutants are generated by applying mutation operators only once) by using the set of 18 mutation operators previously defined for Conceptual Schema based on UML Class Diagram (CD); and, 2) the CoSTest tool (Conceptual Schema Testing) [12] to support the semi-automatic generation of test cases from a requirements model, the execution of CS/CS mutants against generated tests, and reporting the results.

The main contribution of this paper is to empirically evaluate CoSTest's effectiveness in detecting faults and the adequacy of the test suite, using seven CSs and mutation testing.

The paper is organized as follows. Section 2 describes the CoSTest technique. Section 3 presents the experimental design. Section 4 discusses the results. Section 5 summarizes the threats to validity. The conclusions and future work are given in Section 6.

2 AN EARLY TESTING TECHNIQUE: CoSTest

As mentioned in Section 1, the main goal of CoSTest is to automate a testing approach for Conceptual Schemas. For this, CoSTest generates test cases (i.e. assertions with the expected value), transforms the conceptual schema under test into an executable CS and executes the test process for reporting the results. In this section we describe the testing environment, the steps of the CoSTest technique and test cases properties.

2.1 The testing environment

The environment for testing conceptual schemas provided by CoSTest is based on the Action Language for Foundational UML, or Alf [23], adopted as standard by the OMG [25].

Alf is basically a textual notation for UML behaviours that can be attached to a UML model at any point that may contain a UML behaviour, e.g. the method of an operation or the classifier behaviour of a class. As Alf notation includes basic structural modelling constructs, it is also possible to deal with entire models textually in Alf. Semantically, Alf maps the Foundational UML (fUML [24]) subset, then fUML provides the virtual machine for the execution of the Alf language, so that the test suite and executable model are generated and transformed into Alf language, respectively.

2.2 The CoSTest Process

Fig. 1 provides the reader with a description of how CoSTest operates, its phases and activities.

2.2.1 Test suite Generation

1. *Identify the input requirements*: The tester needs to select the requirements model (RM), which is based on Communication Analysis [6]. We assume that the model is syntactically well-formed.
2. *Generate the test model (TM)*: CoSTest analyses the RM structure by automatically traversing all the RM nodes (event sequences) and extracting all the Test Model (TM) elements and their properties.
3. *Generate the abstract test scenarios (TS)*: CoSTest computes the total number of possible test scenarios (based on event sequence) and generates the test scenarios with abstract test cases.
4. *Concretize Variables*: The next step is to concretize the variables of the test cases. The tester can (i) recover a variable list from the test model and generate values automatically from the example values specified in the requirements model, or (ii) concretize manually by introducing values for each variable.
5. *Choose the test suite types*: The tester can select between two types of test cases, such as (i) partial (only positive test cases) ii) complete, which adds test cases with some negative conditions, such as values out of range, constraint violations, and unique value violation for class variables.
6. *Generate concretized test cases (CTC)*: In this phase, CoSTest automatically transforms the abstract test cases into parameterized scripts. The output is a non-executable script for each test scenario. Scripts are not executable in the sense that they do not contain concretized variables.

   CoSTest then computes and generates the total number of possible executable and concrete test cases that may be executed on the CS, including concretized variables, the test objective and an expected output (oracle) that is used to validate the CS requirements. The output of this step is a test suite formed by an executable script (Alf script) for each test scenario. The test suite for the subsequent testing process is now ready.
7. *Identify the Conceptual Schema:* The tester, which is a UML Class Diagram (CD), identifies the Conceptual Schema. We assume that the CS is syntactically well-formed.

2.2.2 CSUT Generation

8. *Generate an Executable Conceptual Schema (CSUT):* CoSTest transforms the CS into an executable format (Alf) for its execution.

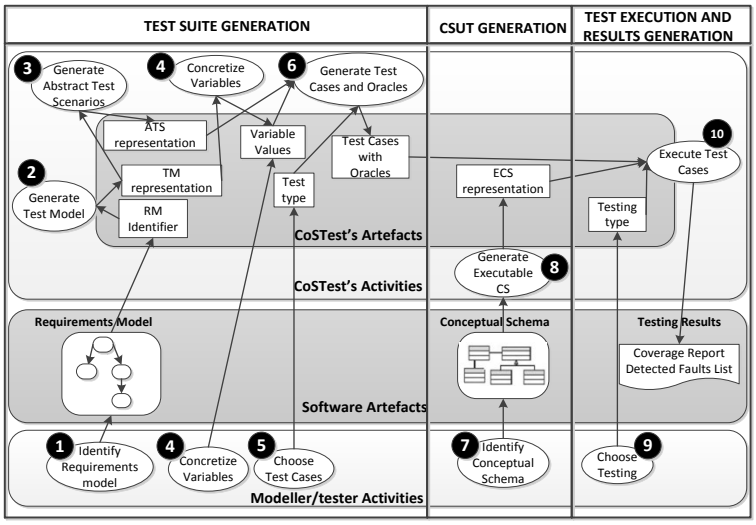

Figure 1. The CoSTest process

2.2.3 Test Execution and Reports Generation

9. *Choose testing type.* The testing type is based on the following stop criteria: Testing should be stopped when (1) one fault is detected; or (2) all available test cases have been run.
10. *Execute Test suites:* Test cases are executed on the executable CS and the output is compared to the stored expected output (from Step 6). CoSTest generates an execution report in which the executed test cases are classified as passed, failed or inconclusive. A coverage analysis is performed and a fault report is generated.

Thus, Fig. 1 contains four main parts: CoSTest artefacts, CoSTest activities, software artefacts and modeller/tester activities. As the names suggest, CoSTest activities are done automatically whereas the modeller/tester activities are done manually. CoSTest encapsulates all the CoSTest artefacts. The numbered ovals represent activities and the boxes represent artefacts. Arrows to/from activities represent the consumption and production of artefacts, respectively.

2.3 CoSTest Test Cases

A test suite for CS is a set of one or more test scenarios. Each test scenario is a story that consists of one or more test cases. The CoSTest test cases exhibit the following properties:

- A test case consists of a fixture and one or more statements that execute one of the tests applicable to CS, such as testing assertions about the occurrence or the non-occurrence of an event. The fixture is a set of statements (e.g. create an object or link, execute a method) that create a CS state and define the values of the CS variables.
- Each execution of a test case starts with the execution of the fixture. For example, if we want to test the creation of an object of the *RegisterUser* class in the *Sudoku Game* CS, a test case that corresponds to a one test scenario generated by CoSTest would be as shown in Fig. 2.
- It is assumed that the execution of each test case starts with an empty state. With this assumption, test cases of a CS are independent of each other, and the order of their execution is therefore irrelevant.

```
private import Sudoku::*;
public import Alf::Library::BasicTypes::*;
public import Alf::Library::Asserts::*;
activity AbsTScenario_1_Sudoku () {
registered_user_=new REGISTERED_USER(p_atrusername="Mafer",
p_atrpassword="password2015",p_atrname="Maria Fernanda",
p_atrsurname="Granda Juca",p_atrmail="mafer.granda@hotmail.com");
AssertTrue("Object Created",registered_user_ instanceof REGISTERED_USER);
```

Fixture

test

Figure 2: A partial view of a test case

- A test case always returns a verdict which may be Pass, Fail or Inconsistent. The execution of the test cases leads to one of the following three outputs:
  - No defects and a status of passed execution. This is considered the output expected.
  - A defect list and a status of failed execution. For example the execution of the test cases may produce an output with several defects (e.g. missing class, incorrect operation and missing operation), which is different from the expected output.
  - A defect list (optional) and "status=inconclusive" if the execution is not conclusive. For example, if the fixture has caused a fault, this leads to an inconclusive status.

In the next section, we describe the design of a controlled experiment for evaluating the proposed technique by means of its effectiveness for detecting faults and test suite quality.

3 EXPERIMENTAL PLAN

Since the experiment was motivated by the need to investigate the effectiveness of our testing tool, we intended to compare the effectiveness and adequacy of the test cases when they were applied in both first order mutants and high order mutants to detect faults in seven CS. The experiment was carried out in 2016

(from January to March) and was designed according to Wholin et al. [29], and reported according to Juristo and Moreno [21]. This section describes the goal of the study, research questions, metrics used, the subject CS, and the experimental settings.

3.1 Goal

In the line with the Goal/Question/Metric Paradigm [27], the goal of our empirical study was the following:

Analyse *the test suite generated by the CoSTest tool* ***for the purpose of*** *carrying out a comparative evaluation* ***with respect to*** *its effectiveness in detecting faults, fault types and the adequacy of the test suite* ***from the point view of*** *the testers* ***in the context of*** *mutants generated for seven CS.*

3.2 Research Questions

As we were interested in determining if the effectiveness was the same for both types of mutants (i.e. FOM and HOM), we posed and studied the following research questions:

- **Q1**: How significant is the influence of the mutation type in CoSTest's effectiveness in detecting faults? As we were also interested in measuring whether the test case quality depends on the type of mutant:
- **Q2**: How adequate are CoSTest test suites for killing both the First Order Mutants and High Order Mutants of Conceptual Schemas?

3.3 Hypotheses

We defined three hypotheses. Table 1 shows the null hypotheses (represented by a 0 in the subscript), which corresponds to the absence of an impact of the independent variables on the dependent variables. The alternative hypotheses involve the existence of such an impact and are the expected result.

Table 1: Specification of hypotheses

Null hypothesis	Statement: Mutant type does not influence …
$H1_0$ (RQ1)	… the effectiveness of the CoSTest test cases in detecting faults in Conceptual Schemas
$H2_0$ (RQ1)	… the effectiveness of the CoSTest test cases in detecting fault types in Conceptual Schemas
$H3_0$ (RQ2)	… the adequacy of the CoSTest test cases

3.4 Variables and Metrics

3.4.1 Independent Variables

We consider one independent variables (a.k.a. factor [21]):

- *Mutation type.* Since this study uses mutation for injecting the artificial faults into a CS, mutants can be classified into two types according to the number of mutated elements:
  - First Order Mutants (FOM), which are generated by applying mutation operators (i.e. rules to modify the grammar used to capture the syntax of a software artefact [18]) only once.
  - Higher Order Mutants (HOM), which are generated by applying mutation operators more than once [18].

3.4.2 Dependent Variables

We consider the following dependent variables (a.k.a. response variables [21]), which are expected to be influenced to some extent by the independent variable.

- *Fault Detection Effectiveness.* To investigate our RQ1 we need to measure the effectiveness of the CoSTest tool in terms of both the number of faults found and the type (or cause) of the faults that were found [22].
- *Adequacy Test Suite.* For a test suite T the adequacy score is a variable that can be used to measure the effectiveness of a test suite in terms of its ability to kill mutants because it is one outcome of the Mutation Testing process, which indicates the quality of the input test set [18].

3.4.3 Effectiveness Metric

For evaluating the effectiveness of our testing technique, we used two metrics:

- *Rate of Fault Detection (FDR).* The metric FDR is the value calculated by dividing the number of faults detected by the tool by the total number of faults that are expected to be identified from the CS mutants.

$$FDR(T) = F_D(T)/F_E \quad (1)$$

- *Rate of Fault Type Detection (FTDR).* The metric FTDR is the value calculated by dividing the number of fault types detected by the tool by the total number of fault types that are expected to be identified from the CS mutants.

$$FTDR(T) = FT_D(T)/FT_E \quad (2)$$

3.4.4 Test Suite Adequacy Metric

During execution each CS mutant *Mi* will be run against a test case suite *T*. If the result of running *Mi* is different from the result of running *CS* for any test case in *T*, then the mutant *Mi* is said to be "killed", otherwise it is said to have "survived". A CS mutant may survive either because it is equivalent to the original model (i.e. it is semantically identical to the original model although syntactically different) or the test set is inadequate to kill the mutant. Thus, the mutation score (MS) for a test suite T is the ratio of the number of killed mutants M_K (T) over the total number of the non-equivalent mutants M_T generated for a CS, as follows::

$$MS(T) = M_k(T)/M_T \quad (3)$$

3.5 Experimental Context

3.5.1 Subject CS

We used seven subject CS in our study which contained a variety of characteristics that can be present in UML CD-based CS, including classes, relations (i.e. association, composite aggregation, and generalization) and different types of constraints (i.e. pre-condition, post-condition and body condition). These CS were of different sizes and domains (e.g. information systems,

games). The subjects included an industrial case (i.e. IM), some others were found in the literature (i.e. [7], [8] and [2]) and others (i.e. ER, OCR and VC) were selected because they contained the CS elements required to inject the faults. Table 2 summarizes the characteristics of these CS.

Table 2: Elements of the Subject Conceptual Schemas

Element	VC	MT	SG	ER	OCR	SS	IM
Classes	5	6	11	7	10	9	6
Attributes	19	26	32	42	62	45	29
Operations	6	13	19	24	16	32	13
Parameters	22	43	48	75	77	91	51
Associations	4	5	11	8	10	9	4
Constraints	17	9	19	21	14	12	8
Generalizations	0	0	4	0	3	0	0

A brief description of each CS is as follows:

1. Video Club (VC) CS represents the functionality of a chain of video stores to manage movies, partners and movie rentals.
2. Medical Treatment (MT) CS defines part of a Medical Treatment business process for a fictional hospital named University Hospital Santiago Grisolía, developed by España et al. [8].
3. Sudoku Game (SG) CS was developed by Tort and Olivé [2] as an object-oriented CS of the Sudoku Game system. This CS defines the functionality for managing different users, playing with their sudokus and generating new ones.
4. Expense Report (ER) CS defines the functionality of an information system to manage the expense-report life cycle of a business. This CS deals with several entities such as departments, employees, projects and expense types.
5. Online Conference Review (OCR) CS, which is based on the description of the CyberChair System [4], defines the functionality of an information system to deal with members (committee chair and program committee) of a conference, as well as authors that submit papers to be evaluated for inclusion in the conference proceedings.
6. Super Stationery (SS) CS defines the information system of a company that provides stationery and office material to its clients. This CS was developed by España et al. [7].
7. Incident Management (IM) CS defines the functionality of an information system to solve the incoming incidents (reception, process, allocation process and resolution process). This CS is a real case taken from Everis Company[1], a multinational firm offering business consulting, as well as development, maintenance and improvement IT.

3.5.2 Mutation operators

In a CS, missing, unnecessary and incorrectly modelled requirements are the main causes of a CS inaccuracy that can be detected by the requirements. In a previous work [12], 50 mutation operators were defined for CS, and 18 were evaluated for generating valid first order mutants. These mutants were generated with the help of a mutation tool prototype (https://staq.dsic.upv.es/webstaq/mutuml.html).

In this work we applied 27 mutation operators out of a total of 50 (see Tables 3-4) to mutate a CS and evaluate CoSTest's effectiveness and the adequacy of the test suite. Table 3 shows 18 mutation operators to create first order mutants and Table 4 lists the 9 mutation operators to create high order mutants.

Table 3: Mutation operators for CS FOM taken from [12]

#	Code	Mutation Operator rule
1	UPA2	Adds an extraneous Parameter to an Operation
2	WCO1	Changes the constraint by deleting the references to a class Attribute
3	WCO3	Change the constraint by deleting the calls to specific operation.
4	WCO4	Changes an arithmetic operator for another and supports binary operators: +, -,*,/
5	WCO5	Changes the constraint by adding the conditional operator "not"
6	WCO6	Changes a conditional operator for another and supports operators: or, and
7	WCO7	Changes the constraint by deleting the conditional operator "not"
8	WCO8	Changes a relational operator for another and supports operators: <, <=, >, >=, ==, !=
9	WCO9	Changes a constraint by deleting a unary arithmetic operator (-).
10	WAS1	Interchanges the members of an Association.
11	WAS2	Changes the association type (i.e. normal, composite).
12	WAS3	Changes the multiplicity of an Association member (i.e. *-*, 0..1-0..1, *-0..1)
13	WCL1	Changes visibility kind of the Class (i.e. private)
14	WOP2	Changes the visibility kind of an operation.
15	WPA	Changes the Parameter data type (i.e. String, Integer, Boolean, Date, Real).
16	MCO	Deletes a constraint (i.e. pre-condition, post-condition constraint, body constraint)
17	MAS	Deletes an Association.
18	MPA	Deletes a Parameter from an Operation.

Table 4: Mutation operators for CS HOM taken from [12]

#	Code	Mutation Operator rule
1	WCO2	Changes the property (attribute) data type in the constraint
2	WGE	Changes the Generalization member ends
3	WAT1	Changes the Attribute feature "Is Derived" to true
4	WAT2	Changes the Attribute property "Is Derived" to false
5	WAT3	Changes the Attribute data type
6	MGE	Deletes a Generalization relation
7	MCL	Deletes the class (i.e. normal or association class)
8	MAT	Deletes an Attribute
9	MOP	Deletes an Operation

Each of the 27 mutant operators is represented by a three-letter acronym and a number. The acronym consists of 3 parts: (i) one letter that corresponds to the defect type injected by the mutation operator (U=unnecessary, W=wrong and M=missing; (ii) two letters that represent the modelling element (i.e. CO=constraint, GE=generalization, AS=association, CL=class, AT=attribute, OP=operation, and PA=parameter) affected by the mutation; and

[1]www.everis.com

(iii) a sequential number within its category, for example, the "Missing Association" (MAS). Fig. 3 shows a partial view of a CS in which five mutation operators have been applied. Four operators generate valid FOM (i.e. b) MPA, c) MCO, d) WCO8, e) WAS3). However, applying the MAS operator to the WhiteCells association generates a non-valid FOM because there is a constraint (i.e. WhiteCells derivation) that is related with the association. Simply deleting the association would result in a Dangling constraint, which evidently is not desirable.Therefore, we need to add more steps to the operator (going from FOM to HOM). The HOM should delete the association together with the respective constraint. This way, the mutant will not be detected by the parser and can generate a valid mutant for testing.
Our experiment was carried out under a within-subject design, all our subjects were exposed to the two treatments of our independent variable (mutation type) [3].

3.6 Experimental Procedure

This section describes the details of the experimental setup including the subject CS used, instrumentation, data collection, and analysis. Fig. 4 summarizes the experimental process, which involved performing the following seven steps:

3.6.1 Choose CS Subjects

The selected subjects are described in Section 3.5.1.

3.6.2 Generate Test Suites

A test suite T was generated to kill CS mutants for each CS subject by following Steps 1-6 of Section 2.2, we then analysed and recorded the information on the generated test cases in order to eliminate repeated or invalid test cases. The CoSTest report was then used for this task

3.6.3 Execute Test Suites on CS

Each test suite is executed on the respective CS subject using our CoSTest tool (https://staq.dsic.upv.es/webstaq/costest.html). We assessed whether an invalid test case required a manual setting (e.g. concretize variables that require several values because they should be unique values or adjust a negative test case so that it can create a valid sequence of events to validate constraints).
We adjusted the test cases in order to get a successful testing process with the original CS and registered the invalid test cases.

3.6.4 Generate CS Mutants

As this step is quite computationally expensive and cumbersome, we used our MµtUML tool [16] for generating first order mutants, in contrast to the high order mutants, which were generated manually. Both mutant types were generated by using the mutation operators introduced in Section 3.5.2. A syntax analysis was then performed by using the Alf parser to ensure that the mutants were valid and could be used in a testing process.
In this study, we used all the FOMs generated by the tool for all CS subjects (see Table 8 in Appendix). In actual testing scenarios, CS do not typically contain as many faults as these numbers of mutants. The numbers of selected mutants derived by this process for our subject CSs can be found at https://staq.dsic.upv.es/webstaq/mutuml/experiment_data.htm.
In the other case, since there is no tool to automatically generate HOMs, to simulate more realistic scenarios, we randomly selected 3 mutants from the pools of mutants created for each mutation operator. Our goal was 27 mutants per CS, 3 mutants by each mutation operator from Table 4, but some versions of our CSs did not have enough mutants to allow formation of so many groups.

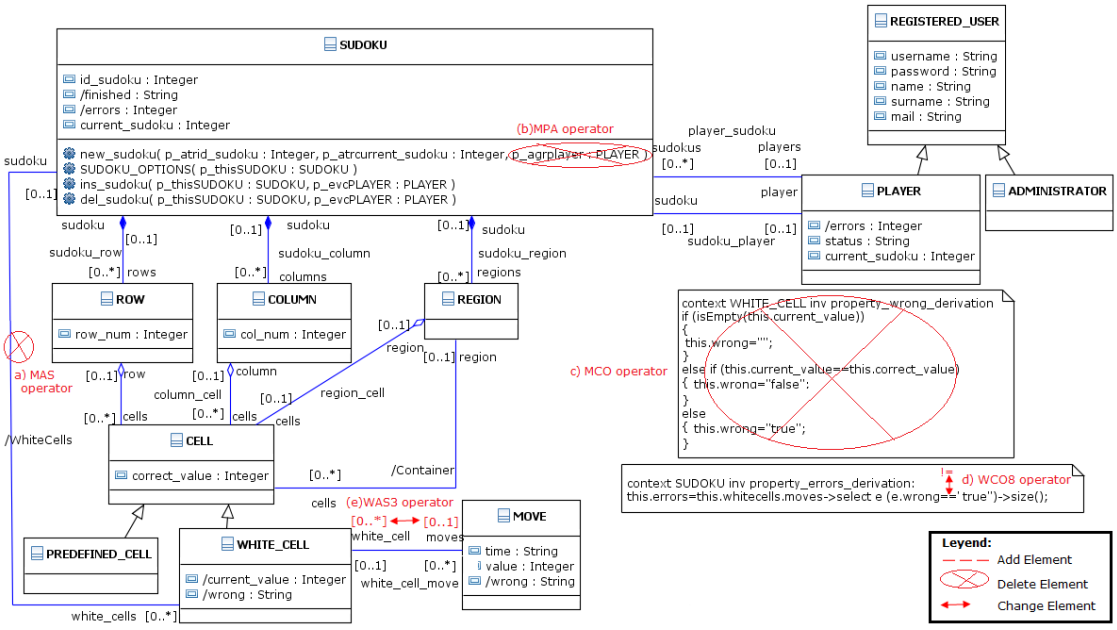

Figure 3. Excerpt of a UML CD-based CS and the application of five mutation operators

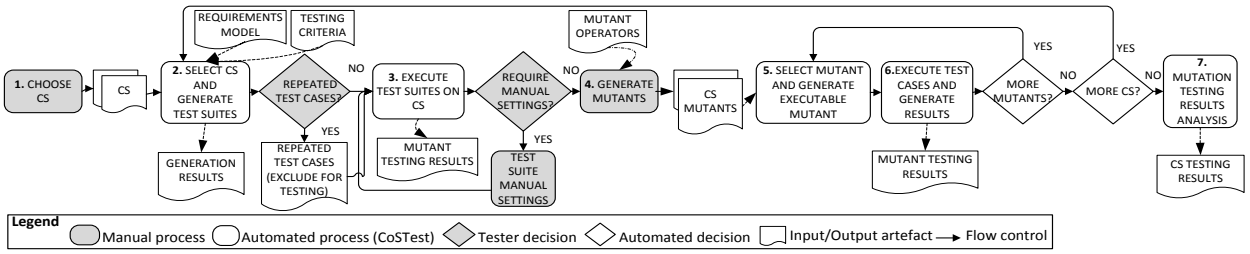

Figure 4. Steps taken in experimental process

So, our random selection algorithm stopped generating mutants for each mutation operator when it could not generate any more unique mutants, resulting in several cases in which mutants numbered less than 27, i.e. for WAT2, WGE and MGE operators (see Table 9 in Appendix).

3.6.5 Select and generate an executable CS mutant

Each CS mutant is transformed into an executable CS (CSUT) by using the respective CoSTest module (see Step 8 in Section 2.2).

3.6.6 Execute Test Suites on CS Mutants

We ran each test case using CoSTest for each mutant and maintained the test status (i.e. passing/failing/inconclusive). We compared the output of each mutant against the output of the original version of the CS with no faults. When the output of the mutant was different to the original CS output, the test case was labelled as failing and when the outputs were exactly the same, the test case was tagged as passing (see Section 2.3). We then manually examined the FOM with zero kills and eliminated any that were semantically equivalent to the original CS. The analysis of survivor mutants in order to identify equivalent mutants is a prerequisite for calculating a mutation score. An example of an equivalent mutant is shown in Fig. 5.

Original Constraint with relational operator "=="

```
this.employees_number=this.employees->select e(e.fired==false)->size()==0? 0:
this.employees->select e(e.fired==false)->size();
```

Mutated Constraint change the relation operator to "<="

```
this.employees_number=this.employees->select e(e.fired==false)->size()<=0? 0:
this.employees->select e(e.fired==false)->size();
```

Figure 5. Excerpt of a Constraint mutated by WCO8

We used the CoSTest option to export the results (faults and coverage analysis) of the testing process of the CS subject. If there are further CS to be studied, Steps 2 to 5 are repeated with the next subject.

3.6.7 Analysis of Testing Results

The CoSTest effectiveness and adequacy of the test suite is calculated from the information recorded in this process. These results are given in the next Section.

4 ANALYSIS AND INTERPRETATION OF RESULTS

This section describes the analysis and interpretation of the results related to our response variables (e) for Q1 and Q2. The Statistical analysis was carried out on the Statistical Package for Social Sciences (SPSS) V23.0.

4.1 Fault Detection Effectiveness

Since the first research question (Q1) was aimed at evaluating CoSTest's effectiveness at detecting faults, we compared the number and types of faults detected for mutant type (i.e. FOM and HOM) in the different CS subjects. Table 5 shows both the number of the faults and the number of fault types detected in each CS subject by mutant type (i.e. FOM and HOM).

Shapiro-Wilk tests were performed to evaluate the samples normality. We used this test as our numerical means of assessing normality because it is more appropriate for small sample sizes (<50 samples).

4.2 Effectiveness based on Rate of Fault Detection

Since all Sig. values for Shapiro-Wilk tests were 0.165 for FOM and 0.001 for HOM, these variables do not follow a normal distribution (<0.05 for HOM).

Table 5: Faults and Fault Types detected by Mutant Type

CS / Fault Types	VC		MT		SG		ER		OCR		SS		IM	
	FOM	HOM	FOM	HOM	FOM	HOM	FOM	HOM	FOM	HOM	FOM	HOM	FOM	HOM
Extraneous Derived Attribute				3						5		3		3
Extraneous Constraint				3	1					2		3		3
Missing Class	5	1	6	3	11	2	7	2	10	2	9	3	6	3
Missing Constraint	52		15		50	10	36	2	37	1	19		21	
Missing Operation		13	7	2	14	4	17	6	6	3	23	4	7	2
Missing Association	4		8						13		12		8	
Incorrect Operation	1	6		9		9		12		13		8	2	9
Incorrect Parameter	3		27	1	29		58	2	16	1	82	1	20	1
FDR	0.71	1.00	0.74	1.00	0.63	0.93	0.71	1.00	0.61	0.90	0.74	1.00	0.58	1.00
FTDR	0.83	1.00	0.80	1.00	0.86	1.00	0.80	1.00	1.00	0.88	0.75	1.00	0.86	1.00

So, we considered both mutant types as independent groups. Then, the Mann-Whitney U Test was used to test our first null hypothesis (H_{10}). Fig. 6 shows the box-plot containing data on the number of faults per mutant type and Table 6 shows the results of the Mann-Whitney U Test.

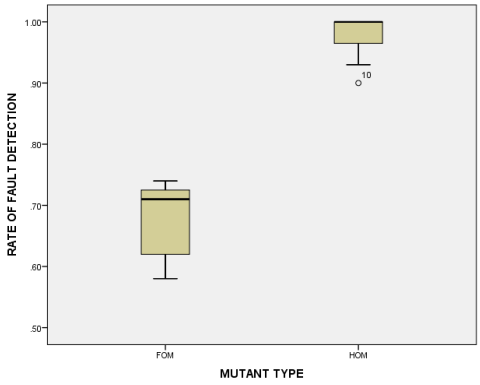

Figure 6. Box-plot for Number of Faults by Mutant Type

From these results, we can see that the HOM group gets higher scores on the dependent variable than the FOM group. Therefore, we rejected hypotheses H_{10}. In other words, "*the rate of fault detection is different for each mutant type; U =0, p=0.001<0.05*".

Table 6: Values of Mann-Whitney U Test

	Rate of Fault Detection
Mann-Whitney U	.000
Wilcoxon W	28.000
Z	-3.209
Asymp. Sig. (2-tailed)	.001

4.2 Effectiveness based on Rate of Fault Type Detection

As in the previous analysis, all Sig. values for Shapiro-Wilk tests were 0.234 for FOM and 0 for HOM, which meant these variables did not have a normal distribution (i.e. <0.05 for HOM). Considering both mutant types as independent groups, we selected the Mann-Whitney U Test (non-parametric test) to evaluate the second null hypothesis (H_{20}). Since the fault type detection rate is different between FOM and HOM (see Fig. 7), we rejected hypothesis H_{20}. In other words, "*the number of fault types detected is different for each mutant type;* (U = 4, p=0.005< 0.05)".

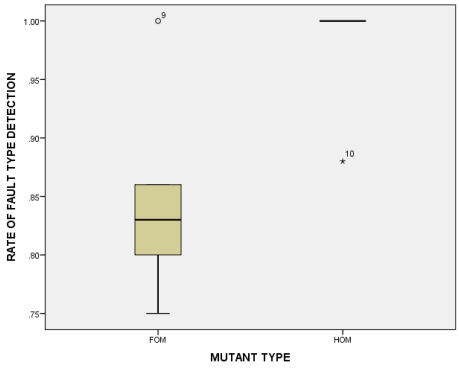

Figure 7. Box-plot for FTDR by Mutant Type

4.3 Test Suite Adequacy

In Q2, we aimed to verify whether the mutation score of CoSTest test suites was the same for killing the different mutant types. To do this, we compared the mutation score for HOMs and FOMs in the seven different CS subjects.

Table 7 shows the mutation score summarized for each CS subject and by each mutant type. Tables 8-9 (see Appendix) show the detailed mutation scores for each CS Subject and mutant type (FOM and HOM) respectively.

Table 7: Mutation Score by Mutant type

Mutant Type	VC	MT	SG	ER	OCR	SS	IM
FOM	0.87	0.80	0.75	0.90	0.75	0.82	0.74
HOM	1.00	1.00	0.89	1.00	0.96	1.00	1.00

Fig. 8 depicts the box-plot of our collected data for mutation score per mutant type. As the results show, the values of mutation score gave a better value for HOM than for FOM.

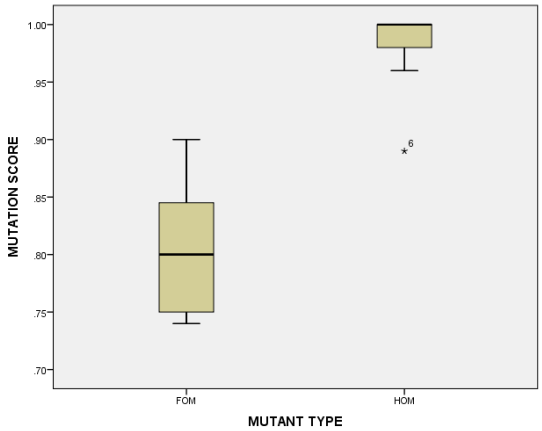

Figure 8. Box-plot-of data for Test Suite Adequacy

As in the analysis (Q1), Shapiro-Wilks tests were performed for each mutant type related to the adequacy of the test suites. Since the value of Sig. for FOM was >0.05 (0.307), this variable had a normal distribution. For HOM the Sig. value was 0, which meant this variable did not have a normal distribution. Considering both mutant types as independent groups, we selected the Mann-Whitney U Test (non-parametric test) to evaluate the hypothesis. From this data, it can be concluded that the mutation score in the HOM group was statistically significantly higher than the FOM group, which meant that we rejected the null hypothesis H_{30} and concluded that "*The test suite adequacy (mutation score) is different for different mutant types;*(U = 1, p=0.002< 0.05)".

4.4 Discussion

Our main results regarding CoSTest's effectiveness and the adequacy of the test suites are the following: mutant type can influence these two variables, with better effectiveness and test suite adequacy in high order mutants than in first order mutants. So, test suites generated by CoSTest are effective at killing a large number of mutants. However, there are fault types that our test suites cannot detect, as explained below.

Thus, the mutants generated by the WAS2 mutation operator (changes the association type, i.e. normal, composite) and WAS3 mutation operator (changes the member end multiplicity of an Association, i.e. *-*, 0..1-0..1, *-0..1) cannot be killed (mutation score=0) by a traditional mutation adequate test set.

Also, the fault types Incorrect Constraint and Incorrect Generalization injected by the mutation operators WCO1, WCO3, WCO4, WCO5, WCO8 and WGE were hard to detect (mutation score <0.7). This showed the weakness of test cases in testing some constraints, such as derivation rules, which needed to be executed in reverse order when there was a relation between classes that affected the computed result. For example, they first calculated the total of the expense report and then the total of the expense report details. This means these test cases will have to be improved.

Additionally, we found that a lower mutation score for some mutants related with constraints (WCOx) was because the test suites only consider coverage at element level and not at constraint level (i.e. condition branch).

We therefore plan to include test cases with values to make sure that different conditions (e.g. > vs >=) will be tested. However, the coverage analysis is important to detect defects when the assertions assert only return values and not side effects (see Fig. 9) in which the coverage analysis is reduced, but all tests still pass.

```
if (number>0) {
    //operation(number); //what if missing (MOP)
    return true;
} else { return false; }
```

Figure 9. Example of an assertion conditional

In addition, we found that CoSTest test suites do not test whether the cardinalities of the association ends meet a certain limit (only creating links according to the test scenario) thereby leading to missed faults, such as an Incorrect Association injected by the WAS3 mutation operator. As well as changing a navigable association to a shared aggregation or vice versa (WAS2) generates an equivalent mutant because "aggregation=shared" has no semantic effect in an executable model using Alf. Thus, another validation technique is required to validate these elements' properties (i.e. inspection of the CS).

Finally, one of the strengths of CoSTest test cases is that it can detect types of defect about misunderstanding requirements (i.e. "Missing" and "Unnecessary" types) that are not normally detected at the CS level, by generating test cases based on user requirements. In a previous work [15] we found a tendency to report only defects related to verification, such as "Wrong" type (e.g. incorrect) rather than defects related to validation.

5 THREATS TO VALIDITY

There are several threats that potentially affect the validity of our study including threats to internal validity, threats to external validity, and threats to construct validity.

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher's knowledge. In our study, the selection of mutation operators is the main threat to internal validity. According to Andrews et al. [1], when using carefully selected mutation operators and after removing equivalent mutants, the mutants can provide a good indication of the fault detection ability of a test suite. Therefore, in order to minimize this threat we used the MµtUML tool [16] to inject faults systematically, by avoiding non-valid and equivalent mutants and optimizing the testing coverage. This tool implements the mutation operators defined in a previous work [12].

Threats to external validity are conditions that limit the ability to generalize the results of our experiments to industrial practice. This threat is reduced by using seven CS of different sizes (see Section 3.5.1) and domain (e.g. information systems, games). Moreover, a CS was taken from industry, some well-documented CS were found in the literature (i.e. [8], [2] and [7]), and others (i.e. ER, OCR, and VC) were selected because they contained the relevant CS elements required to inject the faults.

Threats to construct validity refer to the suitability of our evaluation metrics. We used well-known metrics to measure the effectiveness (rate of number of faults and number of detected fault types) [28] and the adequacy of the test suites (mutation score) [20]. We therefore believe there is little threat to the construct validity.

6 CONCLUSIONS AND FUTURE WORK

Test cases are important artefacts in any software product as a support to users (e.g. modeller/tester/developer) for checking the reliability of their software product.

In this paper, we evaluated empirically the test cases generated by the CoSTest tool with respect to its effectiveness in terms of its fault detection in Conceptual Schemas and the adequacy of the test suite.

Fault detection effectiveness was measured in terms of rate of faults detection and their causes (fault type) by the test suites. Test suite adequacy was measured in terms of the mutation score value. Our evaluation included the analysis of the variables for mutant types (FOM and HOM).

The Effectiveness and adequacy of the test suites was affected by the mutant type and better results were obtained in detecting faults in HOM. These results suggest that the CoSTest technique is robust in detecting types of defects that are not normally detected at the CS level.

However, some mutation operators achieved a value lower than 0.7 in the mutation score. These results suggest that the test suite should include a test for certain characteristics of CS elements, such as associations, and improve the coverage at the constraint level in order to enhance the effectiveness of the test suites.

In future work we plan to identify features of test cases that would lead to improved effectiveness. We also intend to replicate this experiment on a wide variety of subjects to verify the results, including at least two CS (subjects) per domain.

A APPENDIX

Table 8. Mutation Score of CoSTest Test Suites for First Order Mutants

CS / MO	VC			MT			SG			ER			OCR			SS			IM		
	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS
UPA	6	0	1.00	13	0	1.00	19	0	1.00	24	0	1.00	16	0	1.00	32	0	1.00	13	0	1.00
WCO1	0	2	0.00				6	1	0.86	6	3	0.67	1	0	1.00	0	3	0.00			
WCO3	1	0	1.00							4	1	0.80				1	1	0.50			
WCO4	2	0	1.00				7	8	0.54	6	2	0.75				2	0	1.00			
WCO5	1	0	1.00				6	5	0.55	8	3	0.73	6	0	1.00	2	0	1.00	23	0	1.00
WCO6	3	0	1.00				4	7	0.36	2	0	1.00	5	0	1.00	2	0	1.00	20	0	1.00
WCO7							1	0	1.00												
WCO8	40	0	1.00	6	0	1.00	28	13	0.68	20	0	1.00	21	2	0.91	9	4	0.69			
WCO9							1	0	1.00												
WAS1	2	0	1.00	4	0	1.00							7	0	1.00	6	0	1.00	4	0	1.00
WAS2	0	4	0.00	0	5	0.00	0	11	0.00	0	8	0.00	0	10	0.00	0	9	0.00	0	4	0.00
WAS3	0	6	0.00	0	12	0.00							0	21	0.00	0	18	0.00	0	12	0.00
WCL1	5	0	1.00	6	0	1.00	11	0	1.00	7	0	1.00	10	0	1.00	9	0	1.00	6	0	1.00
WOP2	1	0	1.00	7	0	1.00	8	0	1.00	17	0	1.00	6	0	1.00	23	0	1.00	7	0	1.00
WPA	1	0	1.00	9	0	1.00	9	0	1.00	17	0	1.00	3	0	1.00	26	0	1.00			
MCO	15	0	1.00	9	0	1.00	11	0	1.00	15	0	1.00	13	0	1.00	11	0	1.00	0	8	0.00
MAS	2	0	1.00	4	0	1.00							7	0	1.00	6	0	1.00	0	4	0.00
MPA	1	0	1.00	10	0	1.00	11	0	1.00	23	0	1.00	6	0	1.00	32	0	1.00	7	0	1.00
All	80	12	**0.87**	68	17	**0.80**	122	45	**0.74**	149	17	**0.90**	101	33	**0.75**	161	35	**0.82**	80	29	0.74

Table 9. Mutation Score of CoSTest Test Suites for High Order Mutants

CS / MO	VC			MT			SG			ER			OCR			SS			IM		
	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS	K	S	MS
WCO2	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
WGE							1	2	0.33				2	1	0.67						
WAT1	3	0	1.00	3	0	1.00	3	0	0.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
WAT2	2	0	1.00				3	0	1.00	3	0	1.00	1	0	1.00	1	0	1.00			
WAT3	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
MGE							3	0	1.00				3	0	1.00						
MCL	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
MAT	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
MOP	3	0	1.00	3	0	1.00	2	1	0.67	3	0	1.00	3	0	1.00	3	0	1.00	3	0	1.00
All	20	0	**1.00**	18	0	**1.00**	24	3	**0.89**	21	0	**1.00**	24	1	**0.96**	19	0	**1.00**	**18**	**0**	**1.00**

REFERENCES

[1] Andrews, J.H. et al. 2005. Is mutation an appropriate tool for testing experiments? *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (2005), 402–411.

[2] Case Study: Conceptual Modeling of Basic Sudoku: 2006. http://guifre.lsi.upc.edu/Sudoku.pdf.

[3] Charness, G. et al. 2012. Experimental methods: Between-subject and within-subject design. Journal of Economic Behavior and Org.. 81, 1, 1–8.

[4] CyberChair: http://www.borbala.com/cyberchair/.

[5] DeMillo, R. et al. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. Computer. 11, (1978), 34–41.

[6] España, S. et al. 2009. Communication Analysis: A Requirements Engineering Method for Information Systems. 21st International Conference on Advanced Information Systems Engineering (2009), 530–545.

[7] España, S. et al. 2011. Integration of Communication Analysis and the OO-Method: Rules for the manual derivation of the Conceptual Model.

[8] España, S. et al. 2011. Technical Report Communication Analysis and the OO-Method: Manual Derivation of the Conceptual Model the SuperStationery Co. Lab Demo.

[9] Fabbri, S.C.P.F. et al. 1994. Mutation Analysis Testing for Finite State Machines. 5th International Symposium on Software Reliability Engineering (1994), 220–229.

[10] Farooq, U. and Lam, C.P. 2008. Mutation Analysis for the Evaluation of AD Models. International Conference on Computational Intelligence for Modelling Control and Automation, CIMCA. (2008), 296–301.

[11] Ferraz, S. et al. 1999. Mutation Testing Applied to Validate Specifications Based on Statecharts. Software Reliability Engineering, Proceedings. 10th International Symposium on (Boca Raton, FL, 1999), 210–219.

[12] Granda, M.F. et al. 2016. Mutation Operators for UML Class Diagrams. CAiSE 2016 (2016).

[13] Granda, M.F. 2013. Testing-Based Conceptual Schema Validation in a Model-Driven Environment. CAiSE Doctoral Consortium (Valencia, 2013).

[14] Granda, M.F. et al. 2014. Towards the automated generation of abstract test cases from requirements models. 1st Int. Workshop on Requirements Engineering and Testing (Karlskrona, Sweden, Aug. 2014), 39–46.

[15] Granda, M.F. et al. 2015. What do we know about the Defect Types detected in Conceptual Models? IEEE 9th Int. Conference on Research Challenges in Information Science (RCIS) (Athens, Greece, 2015), 96–107.

[16] Granda, M.F. and Condori-fernández, N. 2016. A Model-level Mutation Tool to Support the Assessment of the Test Case Quality. 25TH Int. Conf. on Information Systems Development (ISD2016 POLAND) (2016).

[17] Hamlet, R.G. 1977. Testing Programs with the Aid of a Compiler. IEEE Transactions on Software Engineering. SE-3, 4 (1977), 279 – 290.

[18] Jia, Y. and Harman, M. 2011. An Analysis and Survey of the Development of Mutation Testing. Soft. Engineering, IEEE Transactions on. 37, 5, 1–31.

[19] Jia, Y. and Harman, M. 2009. Higher Order Mutation Testing. Information and Software Technology. 51, 10 (2009), 1379–1393.

[20] Jing, C. et al. 2008. Mutation Testing of Protocol Messages Based on Extended TTCN-3. 22nd International Conference on Advanced Information Networking and Applications (Okinava, Japan, 2008), 667–674.

[21] Juristo, N. and Moreno, A.M. Basics of Soft. Engineering Experimentation.

[22] Morgan, J.A. et al. 1997. Predicting fault detection effectiveness. Proceedings Fourth International Soft. Metrics Symposium (1997), 82–89.

[23] Object Management Group 2013. Action Language for Foundational UML (ALF).

[24] Object Management Group 2012. Semantics of a Foundational Subset for Executable UML Models (fUML).

[25] Object Management Group: www.omg.org.

[26] Pastor, O. and Molina, J.C. 2007. Model-Driven Architecture in Practice. Springer Berlin Heidelberg.

[27] van Solingen, R. and Berghout, E. 1999. The Goal/Question/Metric Method – A Practical Guide for Quality Improvement of Soft. Development. McGraw-Hill.

[28] Vos, T.E.J. et al. 2012. A Methodological Framework for Evaluating Software Testing Techniques and Tools. 2012 12th International Conference on Quality Software. (2012), 230–239.

[29] Wholin, C. et al. 2012. Experimentation in Software Engineering.