A Model-level Mutation Tool to Support the Assessment of the Test Case Quality

Maria Fernanda Granda[1] *fernanda.granda@ucuenca.edu.ec / fgranda@pros.upv.es*
University of Cuenca/ Computer Science Department
Cuenca, Ecuador

Nelly Condori-Fernández *n.condori-fernandez@vu.nl*
Vrije Universiteit van Amsterdam/ Computer Science Department
Amsterdam, The Netherlands

Tanja E. J. Vos *tvos@pros.upv.es*
[1]Universitat Politècnica de València/ PROS Research Center
Valencia, Spain

Oscar Pastor *opastor@pros.upv.es*
Universitat Politècnica de València/ PROS Research Center
Valencia, Spain

Abstract

Although mutation testing is a well-known technique for assessing the quality of tests, there is not a lot of support available for model-level mutation analysis. It is also considered to be expensive due to: (i) the large number of mutants generated; ii) the time-consuming activity of determining equivalent mutants; and (iii) the mutant execution time. It should also be remembered that real software artefacts of appropriate size including real faults are hard to find and prepare appropriately. In this paper we propose a mutation tool to generate valid First Order Mutants (FOM) for Conceptual Schemas (CS) based on UML Class Diagrams and evaluate its effectiveness and efficiency in generating valid and non-equivalent mutants. Our main findings were: 1) FOM mutation operators can be automated to avoiding non-valid mutants (49.1%). 2) Fewer equivalent mutants were generated (7.2%) and 74.3% were reduced by analysing the CS static structure in six subject CSs.
Keywords: Mutation Tool, Model-level Mutation, Class Diagram Mutants, Test Cases Quality.

1. Introduction

In Model-Driven Engineering the models or conceptual schemas (CS) are the primary artefacts in the software development process, and efforts are focused on their creation, testing and evolution at different levels of abstraction. If a model has defects, these are passed on to the following stages of the Software Development Life Cycle, including coding. The quality of a CS can be assessed by detecting its defects during execution. The best test suite is the one that has the best chance of finding defects, but how we do know how good a test suite is? Mutation testing is one of the ways of assessing the quality of a test suite. This method injects artificial faults or changes into a CS (mutant generation) and checks whether a test suite is "good enough" to detect these artificial faults. The artificial faults can be created automatically, using a set of mutation operators (MO) to change (i.e. mutate) some parts of the software artefact. Mutants can be classified into two types: First Order Mutants (FOM) and Higher Order Mutants (HOM) [11]. FOMs are generated by applying mutation operators only once. HOMs are generated by applying mutation operators more than once [10]. However, approaches that employ mutation testing at higher levels of abstraction, especially on CS, are not common [10].

One problem in the design of tests to assess test case quality is that real software artefacts of appropriate size including real faults are hard to find and hard to prepare appropriately (for instance, by preparing correct and faulty versions) [1]. Even when software artefacts with real faults are available, these faults are not usually numerous enough to allow the experimental results to achieve statistical significance [1]. Thus, mutation testing is usually considered expensive due to: (i) the large number of mutants generated; (ii) the time-consuming task of determining equivalent mutants (i.e. functionally identical to the original artefact although syntactically different); and (iii) the time required to compile and execute the mutants [20]. This means mutation testing of real-world software would be extremely difficult without a reliable, fast and automated tool that: (a) generates mutants, (b) runs the mutants against a test suite and (c) reports the mutation score of the test suite.
This paper describes a mutation tool that generates FOMs for CS based on UML Class Diagram (CD) by using previously defined mutation operators [5]. The main usefulness of the mutation tool is to support a well-defined, fault-injecting process to assess the test case quality at the CS level.
The novel contributions of this paper are: 1) the MμtUML prototype mutation tool designed to generate FOMs for UML CD-based CS, eliciting its benefits and weaknesses. 2) An evaluation of the effectiveness and efficiency of the mutation tool to generate valid and non-equivalent FOMs of UML CD-based CS by using six subject CSs.
The rest of this paper is organized as follows. Section 2 describes the background to the study and Section 3 describes the mutation tool itself. The empirical evaluation is described in Section 4. Section 5 presents the results of the evaluation by applying 18 mutation operators to six CSs and a discussion on effectiveness and efficiency of the proposed mutation tool. Section 6 describes possible threats to validity. Section 7 summarizes our conclusions and outlines future work.

2. Background

2.1. Executable Conceptual Schema based on UML Class Diagram

In this paper, defects will be introduced by deliberately changing a UML CD-based CS, resulting in wrong behaviour and possibly causing a failure. As the CS of a system should describe its structure and behaviour (constraints), we represent it by a UML-based (CD). A class diagram is the UML's main building block and shows elements of the system at an abstract level (e.g. class, association class), their properties (owned attributes), relationships (e.g. association and generalization) and operations. In a UML, operations are specified by defining pre- and post-conditions (i.e. constraints) [15]. In this paper we evaluate mutation operators that can inject defects into the following elements: class, attribute, operations, parameters, associations and constraints. In this context, an executable UML model is one with a behavioural specification detailed enough to effectively be run as a program. There are several model execution tools and environments[7]. However, each tool defines its own semantics for model execution, often including a proprietary action language, and models developed in one tool could not be interchanged with or interoperated with models developed in another tool.
In this work, we use the action language adopted as a standard by OMG[8], which is known as the Action Language for Foundational UML, or Alf [13], which is basically a textual notation for UML behaviours that can be attached to a UML model at any point where there is 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 do entire models textually in Alf. Semantically, Alf maps the model to the Foundational UML (fUML [14]) subset, after which fUML provides the virtual machine for the execution of the Alf language.

[7] http://modeling-languages.com/list-of-executable-uml-tools/

[8] http://www.omg.org/

2.2. Mutant Generation Time Estimation Model

The usual process for obtaining values of time on task data involves recruiting users and then performing tests with them in a lab. This procedure, while providing a wealth of informative data can be expensive and time-consuming [16].
Since one of the goals of this study was to analyse the time saved in the mutant generation process by using the proposed tool, we required a method that measured experienced-user task time in order to estimate the time required to generate each mutant type analysed. The most familiar of these cognitive modelling techniques is GOMS (Goals, Operators, Methods and Selection Rules), which has been documented in the still highly referenced text "The Psychology of Human Computer Interaction", by Card, Moran and Newell (1983) [2]. GOMS represents a family of techniques, the most familiar of which is Keystroke-Level Modelling [2]. We selected the Keystroke-Level Model for this study because it has revealed remarkably precise prediction results in several projects such as [8] and [18]. The Keystroke-Level Model predicts the task execution time of a specified interface and task scenario. Basically, it requires a sequence of keystroke-level actions the user must perform to accomplish a task and then adds up the total time required for the actions. The actions are termed at keystroke level if they are actions like pressing keys, moving the mouse, pressing buttons, and so on [12]. The values used for this technique are described in detail in Section 4.2.

3. MutUML: A Mutation Tool

The most critical activity in mutation testing is the suitable design of mutation operators so that they reflect typical defects of the artefact under test. In a previous work [7], we presented a defects classification at model level and in [5] described the process of selection of the 18 mutation operators from a list of 50 for generating First Order Mutants to UML CD-based CS. We developed a mutation tool (https://staq.dsic.upv.es/webstaq/mutuml.html.) for generating first order mutants by using a set of 18 previously defined mutation operators [5], which specify the changes and restrictions required for each mutation operator (see Table 1).

Table 1. Mutation Operators for FOMs taken from [5]

#	Code	Mutation Operator Description
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 operators: <, <=, >, >=, ==, !=
9	WCO9	Changes a constraint by deleting a unary arithmetic operator (-).
10	WAS1	Interchange the members (memberEnd) of an Association.
11	WAS2	Changes the association type (i.e. normal, composite).
12	WAS3	Changes the memberEnd multiplicity of an Association (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.

The tool functionality is separated into the following three processes:

- Calculating Mutants. Testers can select the CS source file (.uml) to calculate the FOMs and also the mutation operators to apply (by default all mutation operators are selected). On pressing the "Calculate Mutants" button, the tool calculates the mutants by applying the mutation operators. The information for each mutant is shown in the "Mutant Description Table" and can be exported as a report by pressing the "Export Report to Excel" button.
- Generating Mutants. The testers/designers can create the mutants required by selecting from the previously calculated mutant list (by default all mutants are selected) and pressing

the "Generate the Mutants" button to generate them. The tool generates the CS mutants (.uml) from the CS source file (.uml).

- Parsing Mutants. After the mutants have been generated they need to be analysed by the parser. This analysis is required before the mutation testing process and also to automatically classify the mutants as valid or non-valid. Each mutant is transformed into an executable format by using Alf language. The Alf parser produces an output with the analysis results of each mutant. To understand how MμtUML works we refer to the partial view of a CS in Fig. 1. Five mutation operators have been applied to the CS. Four operators generate valid FOM (i.e. *b)* UPA2, *c)* WAS3, *d)* WCO3, *e)* MCO). However, applying the MAS operator to the *WhiteCells* association generates a non-valid FOM because there is a constraint (i.e. MovieUnique) 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.

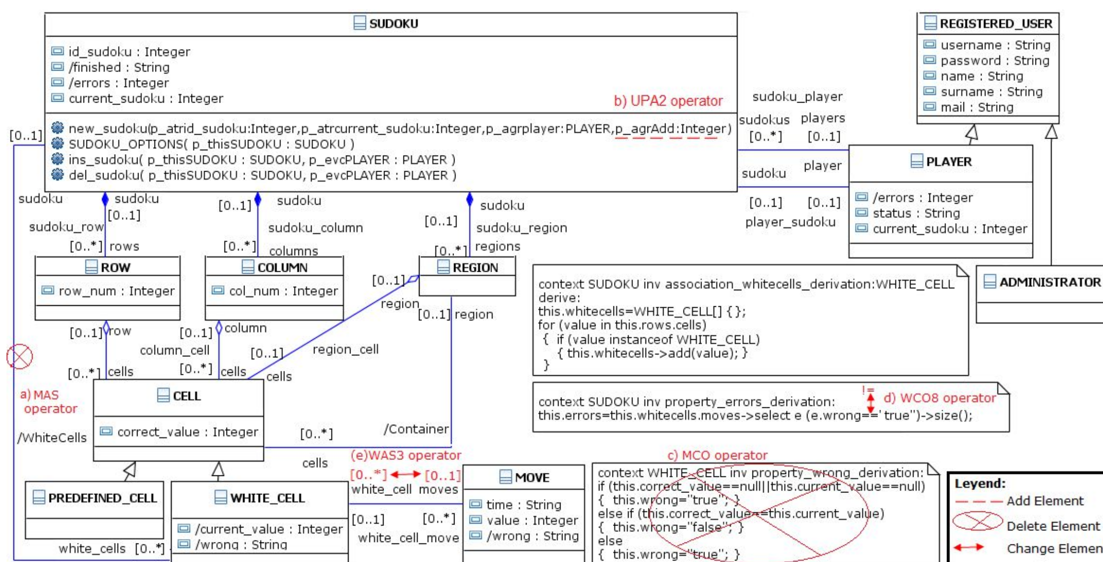

Fig. 1. Excerpt of a UML CD-based CS and the application of five mutation operators, adapted from [19]

4. Empirical Evaluation

This section describes the goal, the research questions, the metrics, the evaluation context and procedure followed for the evaluation.

4.1. Goal

In accordance with the Goal/Question/Metric Paradigm [17] the goal of our empirical study is as follows: **To analyse** the mutant generation strategy of the Mutation tool, **for the purpose of** carrying out an evaluation **with respect to** the effectiveness and efficiency in generating valid First Order Mutants to UML CD-based CS **from the viewpoint of** the researchers.

4.2. Research Questions and Metrics

By means of this study, we aim to be able to respond to the following research questions (RQ):

RQ1: How effective are the mutation operators implemented in a mutation tool for generating FOMs of Conceptual Schema? As this RQ is focused on the generation strategy, the following research questions and metrics are derived from it:

1) RQ1.1. For each defined mutation operator, what is the percentage of valid mutants generated by the mutation tool? The metric M1 for RQ1.1 is the percentage calculated by dividing the number of valid mutants generated by the tool by the total number of mutants that can be generated from the CS elements. The number of mutants that can be generated determines the cost of creating and executing them and also the cost of deriving test cases that kill them. The number of non-valid mutants has an impact on the cost of identifying and discarding them. The mutation tool can indicate whether a mutant is valid or not according to the restrictions defined for each mutation operator.
$$M1(MO) = \frac{M_V(MO)}{M_G(MO)} x\ 100\% \quad (1)$$
2) RQ1.2. For each defined MO, what percentage of parsed mutants is equivalent? The first metric M2 for RQ1.2 is the percentage calculated by dividing the number of valid mutants that are equivalent by the number of valid mutants for mutation operator. The number of equivalent mutants has an impact on the cost of performing mutation testing because a tester needs to execute the test cases against the equivalent mutants to identify and discard them.
$$M2(MO) = \frac{M_E(MO)}{M_V(MO)} x\ 100\% \quad (2)$$
The second metric M3 for RQ1.2 is the percentage calculated by dividing the number of equivalent mutants that can be eliminated using the proposed tool, and total number of equivalent mutants generated by an operator. The cost of performing mutation testing of equivalent mutants can be reduced by the tool by automating an analysis of the subject CS to identify them.
$$M3(MO) = \frac{M_E(MO)\ detected\ by\ CoSTest}{M_E(MO)} x\ 100\% \quad (3)$$

RQ2: To what extent is the generation time reduced by using the tool?

The metric M4 for this RQ is the percentage of time saved by the mutation tool when generating mutants (FOMs). The Manual Generation time is measured by the generation time of valid and non-valid mutants for the subject CS. While the Tool Generation time is calculated by adding the times required to calculate, generate and parse the mutants (FOMs) when using the MμtUML tool. This metric can be measured by applying the following formula:
$$M4(CS) = \frac{Manual\ Generation\ Time(CS) - Tool\ Generation\ Time(CS)}{Manual\ Generation\ Time(CS)} x\ 100\% \quad (4)$$
In order to predict the times for manual generation, the following step-by-step description adapted from Kieras [12] was used to apply the Keystroke-Level Model method in this work.

1) Choose a representative task scenario for each mutation operator. The general scenario required to create manually a CS mutant is: (a) Task1 -open the CS source file, (b) Task2 -duplicate the CS source file, (c) Task3 -select the CS element, (d) Task4 -apply the mutation operator, this task is particular for each mutation operator (see Table 3), and (e) Task 5 -save the mutant and close it.
2) List the keystroke-level actions involved in doing each task with the execution times. The following are some of the standard keystroke-level actions and estimated times for each operator [12].
   - M: Mental operation: User decides or reflects where to click (1.2 sec)
   - H: Home: User moves hand between keyboard and mouse (0.4 sec)
   - P: Point: User point with the mouse to a target on the screen (1.1. sec)
   - K: Set: User clicks on the target (0.28 sec). The time considers the average non-secretarial typist (40 wpm –words per minute) [16].
   - B: Button: User clicks on the button (0.1 sec).
   - BB: User pushes and releases the mouse button rapidly, as in a selection click (0.2 sec).

As we assumed waiting time to be negligible we did not deal with any physical operators for it.

3) List and calculate time for each composite action. Using the granular steps from Keystroke-Level Model, a composite action is clicking on a menu option of the UML diagram editor such as File/Open, Save and Save as, so the four steps are replaced with the composite action: Click on Option. The time to complete this action is modelled in [22]: M (1.2) + P (1.1) + H (0.4) + B (0.1) = approximately 2.8 seconds. Using this method, we defined a small number of composite actions to account for almost all the above five tasks in the 18 mutation operators. The composite actions used were as follows:
   - CA1: Click an Option/Button (MPHB = 2.8 sec).
   - CA2: Double click (MPHBB = 2.9 sec).
   - CA3: Typing Mutant name in a Text Field of the Dialog box. The mutant name is formed by the code of mutant (3 uppercase letters) + an underscore ("_") + a sequential number formed by 3 digits + the file extension ".umlclass". (18K= 5.32 sec). Pressing the SHIFT key counts as a separate keystroke.
   - CA4: Pull-Down List (3.04) (time taken from [16]).
   - CA5: Scrolling (3.96 sec) (time taken from [16]).
4) Estimate the time to complete all scenarios for each mutation operator. For this study we assumed that the person creating the mutants was skilled in: (a) modelling UML CD-based CS, (b) using the UML CD editor (e.g. UML2 tool), and (c) applying the different 18 mutation operators. It was also assumed that the FOMs list had been calculated previously, the UML CD editor had been loaded and active and finally that the tasks were error-free. The Keystroke-Level Model thus addressed only a single aspect of task performance and did not consider other dimensions, such as error-free execution, concentration, fatigue and so on [2]. Using the defined composite action times, the times for each task were calculated (see Tables 7 and 8 in Appendix A).

Table 7 shows the times in seconds estimated by the method for tasks common (i.e. Tasks 1, 2, 3, and 5) to all mutation operators. Table 8 shows the sequence of composite actions required for each mutation operator for Task 4 and the time predicted by Keystroke-Level Model for this task. The time for each mutation operator was estimated in the last column of Table 8 by using the times of the respective tasks for each operator (Task 1 – Task 5).

4.3. Evaluation Context

Subject CSs

Six subject CSs were used in the study (see elements in Table 2). These contained a variety of possible characteristics present in UML CD-based CS, including classes, relationships (i.e. association, composite aggregation, and generalization) and different types of constraints (i.e. pre-condition, post-condition and body condition). Some were found in the literature (i.e. [4], [19] and [3]) and others were selected because they contained the CS elements required to inject the faults.

Table 2. Elements of the subject Conceptual Schemas

Element	MT	SG	ER	OCR	SS	PA
Classes	6	11	7	10	9	15
Attributes	26	26	36	61	44	43
Derived Attributes	0	6	6	1	1	33
Operations	13	19	24	16	32	30
Parameters	43	48	75	77	91	82
Associations	5	6	8	10	9	19
Derived Associations	0	2	0	0	0	0
Composite Aggregations	0	3	0	0	0	0
Constraints	9	19	21	14	12	45
Generalizations	0	4	0	3	0	0

A brief description of each CS is as follows:

1. The Medical Treatment (MT) CS defines part of the CS (of a Medical Treatment business process) of a fictional hospital named University Hospital Santiago Grisolía, developed by España et al. [4].
2. The Sudoku Game (SG) CS was developed by Tort and Olivé [19] as an object-oriented CS of the Sudoku Game system and this CS defines the functionality for managing different users, who play with Sudokus and generating new games.
3. The Expense Report (ER) CS defines the functionality of an information system to manage the expense report life cycle of a business and deals with several entities such as departments, employees, projects and expense types.
4. The Online Conference Review (OCR) CS, which is based on the description of the CyberChair System, 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 a conference to be evaluated for acceptance.
5. The 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. [3].
6. The Photography Agency (PA) CS defines the information system that manages photographers and their photographic reports for distribution to newspaper publishers.

Tools

In this paper one of the aims is to predict the efficiency of the mutation tool in generating valid and non-equivalent mutants in a UML CD editor. The Keystroke Level Model (KLM) Calculator (http://courses. csail.mit.edu/ 6.831/2009/ handouts/ ac18-predictive- evaluation/ klm.shtml) is used for calculating the predictions of task execution times in the UML CD editor from defined scenarios for applying the different mutation operators. This choice is motivated by the large number of publications in the Computer Human Interaction environment using KLM in a variety of emerging application domains [9].
On the other hand, there is no literature available on tools with both integrated functionality (i) the automated generation of test cases and (ii) tests execution for Conceptual Schemas. Therefore, we used our CoSTest CS testing tool (https://staq.dsic.upv.es/webstaq/costest.html) for this work. This tool generates test cases by applying a Model-Driven approach [6]. The test cases use assertions on the return values of the methods and compare them with the post-conditions. We performed the following steps to use the CoSTest tool with UML CD-based CS:

- For each CS, we set the CS testing tool to generate the test cases. We provided the CS testing tool with a requirements model and a set of input values suitable for the subject CS, after which the tool generated the test suite.
- Since a CS is not designed for the CS testing tool, the tool generates an executable CS by applying a transformation from UML to the Alf language.
- The test cases generated by the tool were executed against the CS under test by using the virtual machine for the execution of the Alf language.

A full description of the testing tool is beyond the scope of the present paper.
Finally, while there are tools that support manipulation of UML-based Conceptual Schemas such as Papyrus (http://www.eclipse.org/papyrus/) and UML2 Tools (http://wiki.eclipse.org/MDT-UML2Tools). The UML2 tool is an Eclipse Modelling Framework-based implementation, which is integrated into the tool used for modelling the requirements used as input in the CoSTest tool. For this reason we selected this tool for manipulating UML models.

4.4. Procedure

With the aim of finding empirical evidence to answer the aforementioned RQs, we divided our study into two parts:
The first evaluation was performed to answer RQ1 and partially answer RQ2. In the first evaluation we generated mutants for each mutation operator from the subject CS, then identified

non-valid mutants by applying the MO restrictions and provided suggestions on how to reduce the percentage of equivalent mutants. These results are given in Section 5.1.
The second evaluation was designed to answer RQ2. For this we derived a reliable way to estimate time-on task by using the metric defined in Section 4.2 and the tool summarized in Section 4.3. The generation time savings when using the tool for each subject are given in Section 5.2.

5. Results Analysis

5.1. Effectiveness: Mutant Generation from FOMs Mutation Operators

Table 3 summarizes the results of generating and parsing the mutants for the subject CS by using the mutation tool. For each mutation operator in Table 3 we show the number of (valid and non-valid) possible mutants (MP) that can be generated from the CS elements, the number of mutants generated by the tool (MG) for each of the subject CS, as well as the total number of possible generated mutants and the total number of mutants generated by the tool for all CS. Column M1 shows the percentage of valid mutants generated by the tool from all CS. The highest percentage of M1 (100%) was achieved by implementing the rules and restrictions of the FMOs. It can be seen that the total number of non-valid mutants (1039 or 49.1%) is lower than the valid mutants (1079 or 50.9%). The last cell of the last column (C%) in Table 3 shows the percentage of valid mutants for each mutation by applying the restrictions of the FOMs for all six subject CSs.

Table 3. Generated and Valid Mutants using mutation tool

CS / MO	MT		SG		ER		OCR		SS		PA		All			
	M_P	M_G	M_P	M_G	M_P	M_G	M_P	M_G	M_P	M_G	M_P	M_G	M_P	M_G	*M1 (%)*	*C (%)*
UPA2	13	13	19	19	24	24	16	16	32	32	30	30	134	134	100	100.0
WCO1	0	N/A	7	7	9	9	1	1	3	3	33	33	53	53	100	100.0
WCO3	13	0	19	0	29	5	16	0	34	2	43	13	154	20	100	13.0
WCO4	0	N/A	15	15	8	8	0	N/A	2	2	26	26	51	51	100	100.0
WCO5	0	N/A	11	11	11	11	6	6	2	2	5	5	35	35	100	100.0
WCO6	0	N/A	12	12	2	2	5	5	3	3	4	4	26	26	100	100.0
WCO7	0	N/A	1	1	0	N/A	0	N/A	0	N/A	0	N/A	1	1	100	100.0
WCO8	6	6	47	47	21	21	26	26	13	13	34	34	147	147	100	100.0
WCO9	0	N/A	1	1	0	N/A	0	N/A	0	N/A	0	N/A	1	1	100	100.0
WAS1	5	4	11	0	8	0	10	7	9	6	19	5	62	22	100	35.5
WAS2	5	5	11	11	8	8	10	10	9	9	19	19	62	62	100	100.0
WAS3	15	12	33	0	24	0	30	21	27	18	57	15	186	66	100	35.5
WCL1	6	6	11	11	7	7	10	10	9	9	15	15	58	58	100	100.0
WOP2	13	13	19	19	24	24	16	16	32	32	30	30	134	134	100	100.0
WPA	43	9	48	9	75	17	77	3	91	26	82	12	416	76	100	18.3
MCO	9	9	19	11	21	15	14	13	12	11	45	12	120	71	100	59.2
MAS	5	4	11	0	8	0	10	7	9	6	19	5	62	22	100	35.5
MPA	43	10	48	11	75	23	77	6	91	32	82	18	416	101	100	24.0
All	**176**	**91**	**343**	**185**	**354**	**174**	**324**	**147**	**378**	**206**	**543**	**276**	**2118**	**1079**	**100**	**50.9**

We manually analysed the mutants to determine whether they were equivalent (i.e. the CS mutant produces the same output as the original CS as if it had no faults). The analysed output is produced by the CS testing tool. An example of an equivalent mutant is shown in Fig. 2, where the mutation is not detected by the CS testing tool.

```
Context WHITE_CELL inv property_current_value_derivation:
// Original Constraint with Relational Operator "=="
this.current_value=this.moves->size()==0?-1:this.current_value= this.moves->last().value;
// Mutant Constraint with Relational Operator "<="
this.current_value=this.moves->size()<=0?-1:this.current_value= this.moves->last().value;
```

Fig. 2. Excerpt of a Constraint mutated by WCO8 operator

Table 4 shows the results of analysing equivalent mutants generated in the six CS. For each CS, the table shows the number of equivalent mutants and the percentage of equivalent mutants out of the valid mutants generated by each operator. For example, the first row in Table 4 shows that operator WCO4 had 2 equivalent mutants in the Sudoku CS. These contribute about 13.3% of the 15 valid mutants that the operator has generated. Column M2 in Table 4 shows the percentage of equivalent mutants of the total number of valid mutants for each operator. The

last column in Tables 4 shows the percentage of equivalent mutants generated by each operator out of the total number of equivalent mutants. For example, in the first row of Table 4, the WCO4 operator generated 2 equivalent mutants out of the 78 equivalent mutants that the mutation tool generated in all the CS. It therefore, contributed about 2.6% of the total number of equivalent mutants. The last column in the table shows that most of the equivalent mutants were generated by the WOP2 operator with 74.3% of the total number of equivalent mutants.

Table 4. Number and percentage of equivalent mutants generated using the mutation tool

CS / MO	MT		SG		ER		OCR		SS		PA		All			
	ME	**%**	**ME**	**%**	**ME**	**%**	**ME**	**%**	**ME**	**%**	**ME**	**%**	**ME**	**M2(%)**	**C(%)**	**M3 %**
WCO4			2	13.3									2	3.9	2.6	0
WCO6			1	8.3					1	33.3			2	7.7	2.6	0
WCO8			6	12.8	1	4.8	3	11.5			6	17.6	16	10.9	20.5	0
WOP2	6	46.2	11	57.9	7	29.2	10	62.5	9	28.1	15	50.0	58	43.3	74.3	74.3
All	**6**	**6.6**	**20**	**10.8**	**8**	**4.6**	**13**	**8.8**	**10**	**4.9**	**21**	**7.6**	**78**	**7.2**	**100.0**	**74.3**

We inspected the equivalent mutants to determine why the mutants generated cannot be detected. The reason is that the WOP2 operator (changes the operation visibility) when it is applied on a constructor operation, only affects the access inherited by child classes (a private constructor of the super class is not inheritable). Therefore, it is impossible to detect this mutation operator when the operation is executed in the test cases. A restriction in the rule of the WOP2 mutation operator should be included in the tool to avoid generating this type of mutant. There are other equivalent mutants such as WCO4, WCO6 and WCO8, which can only be identified by inspecting the mutants. The mutation tool cannot avoid producing them. However, by including the above-described implementation restriction for operator WOP2, we see that 74.3% (Metric M3) of the equivalent mutants generated by the mutation tool can be eliminated.

5.2. Efficiency: Generation Time Reduction by Using the Mutation tool

We estimated the time of manual generation of valid and non-valid mutants by using the calculated times for each mutation operator in Section 4.2 (see Table 3).

Table 5 summarizes the times obtained for each mutation operator in each subject CS by generating valid first order mutants. The results show that the subject MT has the lowest mutation time (3661.2 seconds) and subject PA the highest (12516.2 seconds). The last column in Table 5 shows that most time is required to create mutants by using the WCO8 mutation operator (6894.3 seconds), and the shortest time is required to create the mutants by using the WCO7 and WCO9 operators (46.1 seconds). These results are as expected, because these operators generated the highest and lowest values in the number of valid mutants in the six CSs. Some fields in Table 5 are empty because the different subject CS had not the required elements by these mutation operators.

Table 5. Reduced Time in Generating Valid Mutant by Using the Mutation tool

MO	MT (sec)	SG (sec)	ER (sec)	OCR (sec)	SS (sec)	PA (sec)	All (sec)
UPA2	520.0	760.0	960.0	640.0	1280.0	1200.0	5360.0
WCO1		413.3	531.4	59.0	177.1	1948.3	3129.1
WCO3	1018.2		391.6		156.6	1018.2	1566.4
WCO4		695.1	370.7		92.7	1204.8	2363.3
WCO5		475.9	475.9	259.6	86.5	216.3	1514.1
WCO6		596.4	99.4	248.5	149.1	198.8	1292.2
WCO7		46.1					46.1
WCO8	281.4	2204.3	984.9	1219.4	609.7	1594.6	**6894.3**
WCO9		46.1					46.1
WAS1	185.0			323.8	277.6	231.3	1017.7
WAS2	188.1	413.8	301.0	376.2	338.6	714.8	2332.4
WAS3	562.8			984.9	844.2	703.5	3095.4

WCL1	232.7	426.6	271.5	387.8	349.0	581.7	2249.2
WOP2	504.1	736.8	930.7	620.5	1241.0	1163.4	5196.5
WPA	447.7	447.7	845.6	149.2	1293.2	596.9	3780.2
MCO	263.3	321.9	438.9	380.4	321.9	351.1	2077.5
MAS	117.0			204.8	175.6	146.3	643.7
MPA	4359.0	394.9	825.7	215.4	1148.8	646.2	3590.0
All	**3661.2**	7978.7	7427.2	6069.5	8541.5	**12516.2**	46194.3

Fig. 3 shows the total times that are reduced by avoiding the manual generation of valid (i.e. 46194.3 seconds) and non-valid mutants (i.e. 48833.4 seconds) in the six subject CSs.

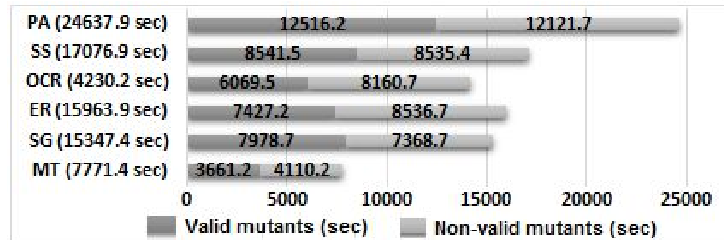

Fig. 3. Time required by a manual generation of valid and non-valid mutants

From this chart we can see that the generation time depends on number of mutants and the time required by each applied MO. For example, in subject MT, the number of valid mutants (91) is higher than non-valid mutants (85) (see Table 3). However, the generation time of non-valid mutant is the longer time (see Figure 3). This result is because the some mutation operators for generating non-valid mutants (e.g. WCO3, WPA) require a longer time (see Table 3 and Table 8).

Additionally, we estimated the time needed for manually creating equivalent mutants by using the WOP2 operator (58 equivalent mutants * 38.78 second/WOP2 mutant =2249.24 seconds), which can be avoided by implementing the suggestion described in Section 5.1 in the mutation tool.

Finally, Table 6 shows the time required to calculate, generate and parse the mutants by using our mutation tool for the different subject CS in this study. The results show that the calculation time is negligible when using the tool, while manual generation time is the longest. The time percentage can thus be reduced by more than 92% in the six evaluated CSs by using the proposed mutation tool.

Table 6. Time used by the Mutation Tool for generating mutants (FOM) in the subject CS

Task Time	MT	SG	ER	OCR	SS	PA
Using the tool (sec)	527.047	1165.098	1053.086	585.069	808.071	1506.162
Reduced Time (sec)	7244.353	14182.302	14910.814	13645.131	16268.829	23131.738
Reduced % (M4)	93.2 %	92.4 %	93.4 %	95.9 %	95.3 %	93.9 %

6. Threats to Validity and Limitations

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. In order to minimize this threat we used a set of 18 previously defined mutation operators [5] to inject faults systematically.

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 six CSs of different sizes (see Section 4.2) and domains (e.g. information systems, games). Some well-documented CS were

found in the literature (i.e. [4], [19] and [3]), and others (i.e. ER, OCR and PA) 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 (number of valid and equivalent mutants) and efficiency (time needed to generate the mutants). In order to perform a specific quantitative analysis for the time saved in the FOM generation process by using the mutation tool. We estimated the time that a user needs to perform the task manually using the Keystroke-Level Model [2]. This model appears to us simple, accurate, and flexible enough to be applied in evaluation situations like ours. However, the model addresses only a single aspect of task performance and does not consider other dimensions, such as error-free execution, concentration, fatigue and so on [2]. Also, this model has several restrictions (e.g. the user must be an expert; the method must be specified in detail; and the performance must be error-free). However, we believe that this model represents an appropriate estimation of the time saved by using the mutation tool and that there is hence little threat to the construct validity.

7. Conclusions and Future Work

Mutations applied at the model level can improve early development of high quality test suites and can contribute to developing high quality systems, especially in a model-driven context. In this paper, we propose a tool that automates the generation of mutants for UML CD-based CS by using a set of previously defined mutation operators. This tool was evaluated for its effectiveness and efficiency in terms of its percentage of valid and non-equivalent mutants and the time that can be saved by using it.

The results show that the mutation operators can be automated avoiding the generation of a high percentage (49.1%) of non-valid mutants. Thus, the tool generates a low percentage (7.2%) of equivalent mutants. However, detecting these mutants is costly in terms of the time and effort of creating, executing and manually inspecting them. We therefore implemented the restrictions and rules for eliminating them by performing a static analysis of the CS. As these results show, the reduction achieved in this analysis of equivalent mutants is about 74.3%, which is equivalent to 2249.24 seconds estimated by KLM, and the cost of reducing non-valid mutant is 49.1% (48833.4 seconds estimated by KLM) by using the mutation tool in the six subject CSs involved in this study. Therefore, the results of this study suggest that the mutation tool can help researchers and supports a well-defined, fault-injecting process to generate a potentially large number of valid and non-equivalent FOMs, increasing the statistical significance of results obtained in assessing test case quality.

This study is a part of a more extensive research project, whose main goal is to propose an approach for testing-based conceptual schema validation in a Model-Driven Environment. We have identified three directions in which to extend this work. First, we intend to study the use of HOMs and subsuming HOMs for UML CD-based CS in order to cover all CS elements and other types of faults in UML CD-based models. Secondly, we hope to evaluate the use of HOMs and compare them with FOMs in order to reduce the cost of mutation analysis. Finally, we plan to perform a large-scale empirical study on several industrial subject CS in order to evaluate the effectiveness of the automatized mutation operators in the mutation tool.

Acknowledgments

This work has been developed with the financial support by SENESCYT of the Republic of Ecuador, SHIP (SMEs and HEIs in Innovation Partnerships, ref: EACEA/A2/UHB/CL 554187), PERTEST (TIN2013-46928-C3-1-R), European Commission (CaaS project) and Generalitat Valenciana (PROMETEOII/2014/039).

Appendix A

Table 7. Estimated Time by Keystroke-Level Model

Tasks	Operator sequence	Time (sec)
Task 1	CA1 (Open option)	2.9
Task 2	CA1 (File option) + CA1 (Save As option) + CA3 + CA1 (Ok button)	13.72
Task 3	CA5 + CA1	6.76
Task 5	CA1 (Save option) + CA1 (Close button)	5.6
All four tasks		28.98

Table 8. KLM Estimation for Task 4 by each Mutation Operator

MO	Task 4 Scenarios for each mutation operator	Task4 time (sec)	Tasks 1-5 time (sec)
UPA2	CA2 (select operation) + CA1 (locate the place to edit the parameter)+18K (Parameter: String) + 1K <enter>	11.02	40.00
WCO1	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA2 (select attribute) + K (<supr> press) + 4K (<shift>) + K (“_”) + 4K (var_auxi)+ CA1() + 4K (<shift>) + K (“_”) + 5K (<shift>) + 6K (<shift>) 5K (<shift>) + 7K(<shift>) + K + CA1 (add button) + CA1 (Ok button)	30.6	59.04
WCO3	CA2 (select constraint) + CA1 (locate the place to edit a variable)+23K (e.g. var_auxi=new Real(0,0);) + CA1 (locate the place to replace the operation for variable)+CA2(select operation)+ 8K (var_auxi) +1K <enter>	20.36	78.32
WCO4	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA1 (text point) + K (<supr>press) + K (write operator) + CA1(add button) + CA1 (OK)	17.36	46.34
WCO5	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA1 (text point) + K (write operator) + CA1(add button) + CA1 (OK)	14.28	43.26
WCO6	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA1 (text point) + K (<supr> press) + K (<supr> press) + K (write operator) + K (write operator) + CA1(add button) + CA1 (OK)	19	49.70
WCO7	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA1 (text point) + K (<supr> press) + CA1(add button) + CA1 (OK)	17.08	46.06
WCO8	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA1 (text point) + K (<supr> press) + K (<supr> press) + K (write operator) + K (write operator) + CA1(add button) + CA1 (OK)	17.92	46.90
WCO9	CA1 (body property) + CA1(edit button) + CA1 (remove button) + CA1 (text point) + K (<supr> press) + CA1(add button) + CA1 (OK button)	17.08	46.06
WAS1	CA1 (Source End) + CA1 (Type property) + CA4 (select type) + CA1 (Target End) + CA1 (Type property) + CA4 (select type)	17.28	46.26
WAS2	CA1 (Target End) + CA1 (Aggregation property) + CA4 (select aggregation type)	8.64	37.62
WAS3	CA1 (Source End) + CA1 (select Lower) + K (new value) + CA1 (select Upper) + K (new value) + CA1 (Target End) + CA1 (select Lower) + K (new value) + CA1 (select Upper) + K (new value)	19	46.90
WCL1	CA5 (Class properties) + CA1 (Visibility property) + CA4 (select visibility)	9.8	38.78
WOP2	CA5 (Operation properties) + CA1 (Visibility property) + CA4 (select visibility)	9.98	38.78
WPA	CA1 (right button) + CA1 (parameters manage) + CA1 (select data type) + CA1 (select edit button) + CA5 (data types) + CA1 (OK button) + CA1 (OK button of parameters manage)	21.84	49.74
MCO	K (<supr> press)	0.28	29.26
MAS	K (<supr> press)	0.28	29.26
MPA	K (F2 to edit operation) + CA2 (select the attribute) + k (<supr> press) + CA2 (select the data type) + k (<supr> press) + k (<supr> press on “:”)	6.92	35.90

References

1. Andrews, J.H. et al.: Is mutation an appropriate tool for testing experiments? In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. pp. 402–411 (2005).
2. Card, S.K. et al.: The keystroke-level model for user performance time with interactive systems. Commun. ACM. 396–410 (1980).
3. España, S. et al.: Integration of Communication Analysis and the OO-Method: Rules for the manual derivation of the Conceptual Model. , Valencia (2011).
4. España, S. et al.: Technical Report Communication Analysis and the OO-Method : Manual Derivation of the Conceptual Model the SuperStationery Co. Lab Demo. , Valencia (2011).

5. Granda, M.F. et al.: Mutation Operators for UML Class Diagrams. In: CAiSE 2016. (2016).
6. Granda, M.F. et al.: Towards the automated generation of abstract test cases from requirements models. In: 1st International Workshop on Requirements Engineering and Testing. pp. 39–46 IEEE, Karlskrona, Sweden (2014).
7. Granda, M.F. et al.: What do we know about the Defect Types detected in Conceptual Models ? In: IEEE 9th Int. Conference on Research Challenges in Information Science (RCIS). pp. 96–107 IEEE, Athens, Greece (2015).
8. Haunold, P., Kuhn, W.: A keystroke level analysis of a graphics application: manual map digitizing. In: CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 337–343 (1994).
9. Holleis, P. et al.: Keystroke-level model for advanced mobile phone interaction. CHI '07 Proc. SIGCHI Conf. Hum. factors Comput. Syst. 1505–1514 (2007).
10. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. Softw. Eng. IEEE Trans. 37, 5, 1–31 (2011).
11. Jia, Y., Harman, M.: Higher Order Mutation Testing. Inf. Softw. Technol. 51, 10, 1379–1393 (2009).
12. Kieras, D.: Using the keystroke-level model to estimate execution times. (2001).
13. Object Management Group: Action Language for Foundational UML (ALF). (2013).
14. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML). (2012).
15. Object Management Group: Unified Modeling Language (UML). (2015).
16. Sauro, J.: Estimating Productivity: Composite Operators for Keystroke Level Modeling. Human-Computer Interact. New Trends. 1–10 (2009).
17. van Solingen, R., Berghout, E.: The Goal/Question/Metric Method – A Practical Guide for Quality Improvement of Software Development. McGraw-Hill Publishing Company (1999).
18. Teo, L., John, B.E.: Comparisons of keystroke-level model predictions to observed data. In: CHI '06: CHI '06 extended abstracts on Human factors in computing systems. pp. 1421–1426 (2006).
19. Tort, A., Olivé, A.: Case Study: Conceptual Modeling of Basic Sudoku, http://guifre.lsi.upc.edu/Sudoku.pdf.
20. Vincenzi, A.M.R. et al.: Muta-Pro: towards the definition of a mutation testing process. J. Brazilian Comput. Soc. 12, 2, 49–61 (2006).