Mutation Operators for UML Class Diagrams

Maria Fernanda Granda[1, 3], Nelly Condori-Fernández[2], Tanja E. J. Vos[3] and Oscar Pastor[3]

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

[2] Vrije Universiteit van Amsterdam, Amsterdam, The Netherlands
n.condori-fernandez@vu.nl

[3] Universitat Politècnica de València, PROS Research Centre,Valencia, Spain
{fgranda, tvos, opastor}@pros.upv.es

Abstract. Mutation Testing is a well-established technique for assessing the quality of test cases by checking how well they detect faults injected into a software artefact (mutant). Using this technique, the most critical activity is the adequate design of mutation operators so that they reflect typical defects of the artefact under test. This paper presents the design of a set of mutation operators for Conceptual Schemas (CS) based on UML Class Diagrams (CD). In this paper, the operators are defined in accordance with an existing defects classification for UML CS and relevant elements identified from the UML-CD meta-model. The operators are subsequently used to generate first order mutants for a CS under test. Finally, in order to analyse the usefulness of the mutation operators, we measure some basic characteristics of mutation operators with three different CSs under test.

Keywords: mutation testing, mutation operators, test cases quality, conceptual schemas, class diagram.

1 Introduction

A conceptual schema (CS) defines the general knowledge required by an information system in order to perform its functions [1], so that an accurate representation of this information (following the requirements) is a key factor in the successful development of the system, especially in a Model-driven environment context [2]. The development of a conceptual schema is an iterative process involving evaluation of the CS, its accuracy and its improvement from the evaluation results. Testing is a well-established technique that helps to accomplish this task. It provides a level of confidence in the end product based on the coverage of the requirements achieved by the test cases.

In this context, we proposed an approach for testing-based validation of Object-Oriented Conceptual Schemas in a Model-driven environment [3][4], where one group of engineers (e.g. requirements engineers) specifies requirement models (RM) from which the test scenarios with test cases (i.e. an executable concrete story of a user-system interaction and the expected result) are automatically generated. These test cases are then used to test the conceptual schemas in an early phase of software analysis and design. Since testing is performed to provide insight into the accuracy of a CS, we need to ensure the test suite quality (i.e. ability to reveal faults).

Mutation testing assesses the quality of a test suite [5] using mutation operators to introduce small modifications or mutations into the software artefact under test, e.g.

adfa, p. 1, 2011.

CS. The artificial faults can be created using a set of mutation operators to change ("mutate") some parts of the software artefact. Mutants can be classified into two types: First Order Mutants (FOM) and Higher Order Mutants (HOM) [6]. FOMs are generated by applying mutation operators only once. HOMs are generated by applying mutation operators more than once [5]. Assuming that the software artefact being mutated is syntactically correct, a mutation operator must produce a mutant that is also syntactically correct. Each faulty artefact version, or mutant, is executed against the test suite. The ratio of detected mutants is known as the "mutation score" and indicates how effective the tests are in terms of fault detection. Approaches that employ mutation testing at higher levels of abstraction, especially on CS, are not common [5].

In Mutation testing the most critical activity is the adequate design of mutation operators so that they reflect the typical defects of the artefact under test. This paper presents the design of a set of mutation operators for Conceptual Schemas (CS) based on Unified Modelling Language (UML) Class Diagrams [7]. The main potential advantage of mutation operators is to describe precisely the mutants that can generate and thus support a well-defined, fault-injecting process [8]. The main contributions of this paper are:

- It provides a classification of 50 mutation operators for UML CD-based CS, which may be used in evaluating verification[1] and validation[2] approaches. The resulting operators are mainly based on a defects classification reported previously [9].
- It illustrates the application of an effective subset of 18 mutation operators, which generate only first order mutants. These mutation operators were applied to three UML CD-based CS with the aim of showing their usefulness in evaluating testing approaches.

The paper is organized as follows. Section 2 describes an UML CD-based CS. Section 3 reviews the defect types at the model level. Section 4 explains the design process of the mutation operators. Section 5 demonstrates the application of the operators in three CS. Section 6 summarizes related work. Finally Section 7 concludes.

2 UML CD-based Conceptual Schemas

The aim of this work is to design mutation operators for evaluating the effectiveness of test cases in finding faults in a CS during the analysis and design of the software. The defects will be introduced by deliberately changing a UML CD-based CS, resulting in wrong behaviour possibly causing a failure.

The CS of a system should describe its structure and its behaviour (operations). In this paper a UML-based class diagram is used to represent such a CS. A class diagram

[1] Verification is to check that the conceptual schema meets its stated functional and non-functional requirements (making the right product) [27].

[2] Validation is to ensure that the conceptual schema meets the customer's expectations (making the product right) [27].

(see Fig. 1) is the UML's main building block that shows elements of the system at an abstract level (e.g. class, association class), their properties (ownedAttribute), relationships (e.g. association and generalization) and operations. In UML an operation is specified by defining pre- and post-conditions. Fig. 1 shows an excerpt of the UML structure for a class diagram and highlights eight elements of interest for this work. Finally, mutation testing requires an executable CS for validating the behavioural aspects included in the CS structural elements. Therefore, we used the Action Language for Foundational UML (Alf [10]) and the virtual machine of Foundational UML (fUML [11]) as the execution environment for mutation testing.

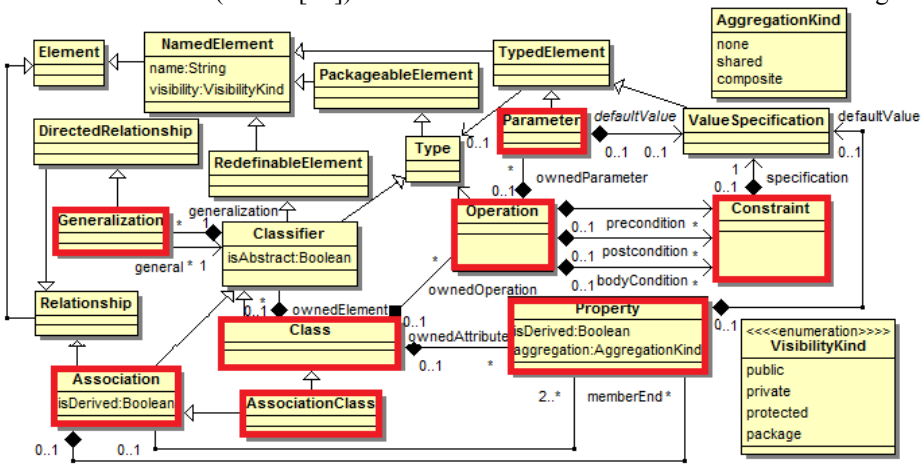

Fig. 1. Excerpt of the Meta-model of an UML Class Diagram [7]

3 Defect Types in UML-based Conceptual Schemas

An important aspect when applying mutation testing to a CS is that the injected defect should represent common modelling errors. In previous work [5] we classified UML model defects reported in the literature and related the types of the defects with the CS quality goals affected by them. Table 1 summarizes the defect types for CS.

Table 1. Defect types in a UML-based model (excerpt taken from [9])

Defect Cause	Sub modes
MISSING Something is absent that should be present.	
WRONG Something is incorrect, inconsistent or ambiguous.	***Inconsistent***: There are contradictions in the models (1) vertical inconsistency (i.e. contradictions between model versions) and (2) horizontal inconsistency (i.e. contradictions between different model views).
	Incorrect: There is a misrepresentation of modelling concepts, their attributes and their relationships, as well as the violation of the rules by combining of these concepts at the time of building partial or complete models.
	Ambiguous (wrong wording): The representation of a concept in the model is unclear, and could cause a user (modeller, low-level designer, etc.) to misinterpret or misunderstand the meaning of the concept.
UNNECESSARY (Extra) Something is present that need not be.	***Redundant***: If an element has the same meaning that other element in the model.
	Extraneous: If there are items that should not be included in the model because they belong to another level of abstraction, e.g. details of implementation, which are decisions (e.g. type of data structure used at code level) that are left to be made by the developers, and is not specified at an earlier level (e.g. CS).

Missing and unnecessary elements (i.e. redundant and extraneous) and incorrectly modelled requirements are the main causes of a design model inaccuracy that can be detected basing on requirement testing. Inconsistency defects require comparing CS versions in order to find them. Finally, ambiguous elements require of user (e.g. modeller, developer) criteria for finding defects.

4 Design of Mutation Operators

As can be seen in Fig. 2, a *CS mutant* M_i is a faulty CS, which is generated by injecting *defects* (adding, deleting or changing elements) into modelling elements (see Fig. 1 in section 2.1) of the original CS. A transformation rule that generates a mutant from the original model is known as a mutation *operator*. If the mutant is generated by applying only one mutation operator in the original CS, it is a first order mutant (e.g. CS with an added constraint), otherwise, it is a higher order mutant if it applies various changes in the CS by using nested operators. For example, a CS that has been mutated by deleting a class has also evidently deleted associations, properties, constraints, operations and parameters associated with the deleted class.

During execution each CS mutant M_i will be run against a test case suite *T*. If the result of running M_i is different from the result of running *CS* for any test case in *T*, then the mutant M_i 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.

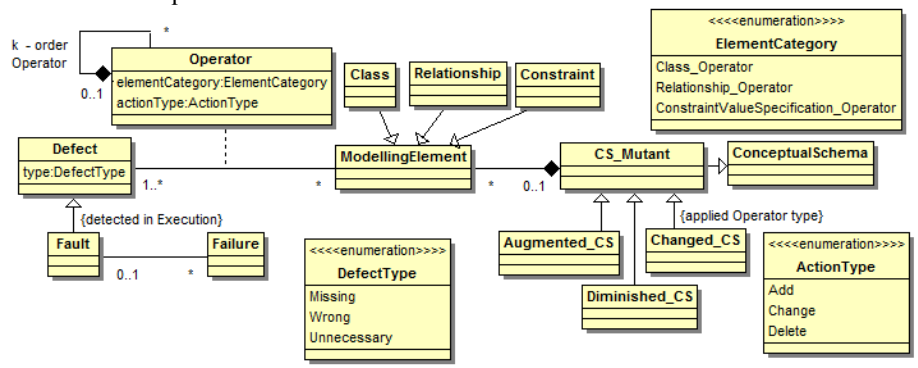

Fig. 2. Relationships among conceptual entities used in the mutant definition (adapted from [12])

To apply Mutation Analysis in the context of UML CD-based CS we need to formulate mutation operators for CS. Mutation is based on two fundamental hypotheses, namely, the *Competent Programmer Hypothesis* (CPH) and the *Coupling Effect Hypothesis* (CEH), both introduced by DeMillo et al. [13]. The CPH states that a program produced by a competent programmer is either correct or near the correct version. The CEH states that complex (or higher-order) mutants are coupled to simple mutants in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults [14]. Consequently, we use the following guiding principles [15]:

– Mutation categories should model potential faults.
– Only syntactically correct mutants should be generated
– Only first-order mutants should be generated

4.1 Mutation Operators Categories

There are several elements of a CS that can be subject to faults. The defined mutation operator set takes the intrinsic characteristics of a UML CD-based CS into consideration, where some UML elements are composed by other elements. They are thus divided into **seven categories**: (1) constraint operators, (2) association operators, (3) generalization, (4) class operators, (5) attribute operators, (6) operation operators, and (7) parameters operators. Each element-based group is then sub classified according to the **three defect types of UML models** (i.e. unnecessary, wrong or missing) [9]. However, as our research focuses on defining mutation operators for evaluating testing approaches, the inconsistent and ambiguity defects are not addressed in this work because they generate a faulty CS that is detected without requiring execution (i.e. testing is not required). The faulty CS is not detected by comparing the model against the requirements. Inconsistency defects are detected by comparing models to detect contradictions between them. Ambiguity defect are detected by the modeller which finds that the representation of a concept in the model is unclear. So that **twenty-one** categories are obtained, such as Unnecessary Constraint (UCO), Wrong Constraint (WCO), Missing Constraint (MCO), Unnecessary Association (UAS), Wrong Association (WAS); Missing Association (MAS) and so on.

Based on the UML meta-model (see Fig 1) and the defects and faults reported in the literature [9], [16], [17], [18], we identified CD element features that can be mutated for their usefulness in evaluating testing approaches:

- Mutating Classes: The attributes isAbstract and Visibility can be mutated.
- Mutating Class Attributes (i.e. Class Variables): The visibility, isDerived, and data type of the variables can be mutated.
- Mutating Operations: The visibility and returned value type when the operation isQuery can be changed. Additionally, swapping compatible parameters in the definition of an operation can be another operation mutant.
- Mutating Parameters: The data type can be mutated.
- Mutating Associations: The visibility, isDerived can be mutated. Additionally, swapping the member of the Association, the kind aggregation and multiplicity for the members of the Association can be mutated.
- Mutating Generalization: swapping the member of the Generalization.
- Mutating Constraints: Changes the constraints by mutating operators (arithmetic, conditional, and negation), references to class attributes, references to operations.

These categories and the main element features give rise to 50 mutation operators (see Table 4 in Appendix). Each of the 50 mutant operators is represented by a three-letter acronym of its category and a sequential number within its category if it is necessary. Some of these operators resulted in a CS that is determined to be faulty without requiring execution (i.e. testing is not required) and others resulted in

behavioural faults (i.e. testing is required). Some of them generate FOM and others HOM. Since we only focus in FOM, 18 mutation operators (see the mutation operators marked with "*" in Table 4) that can generate FOM were obtained through two iterations, as follows (see Figure 3).

First iteration (Exclude equivalent and non-valid mutants): We obtained a detailed list of actions that involve applying each mutation operator, to obtain the rules for each mutation operator (see Table 4).

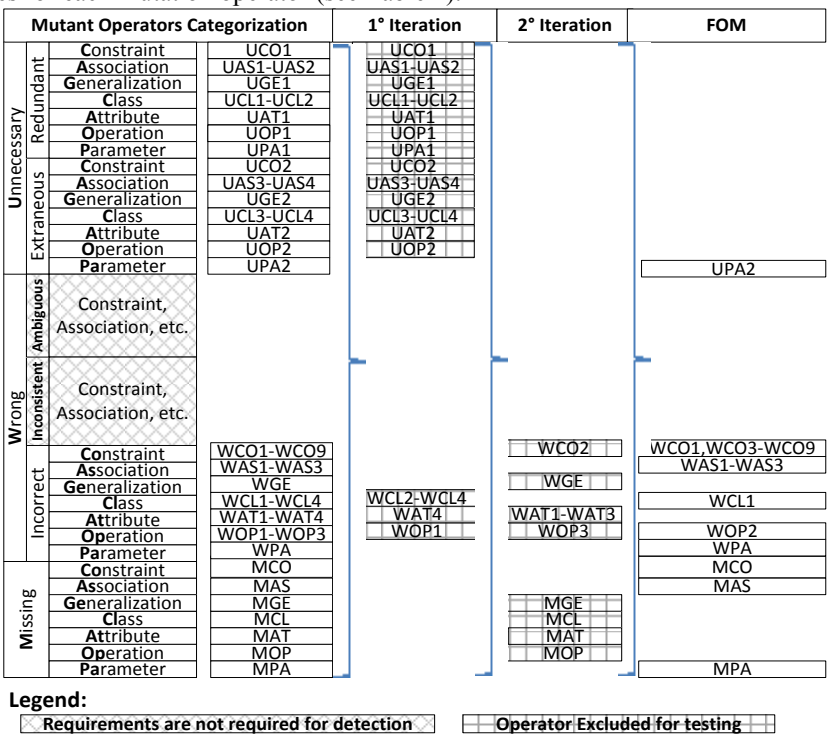

Fig. 3. Selection process of the mutation operators used for evaluating testing approaches

If the rule to generate the mutant is not followed, the mutant generated is a non-valid mutant, which can be detected at parser level. For example, the mutation operator MAS causes an association in a CS to be deleted, however, the constraints related with this association must be deleted or changed in order to generate a valid mutant, otherwise this mutant will be detected by the parser and cannot be used for a testing process. We analysed the mutation operators that always generate a non-valid or equivalent mutant. These results are included in Table 4 as a restriction in the operator rule. These mutation operators are described as follows:

- Adding duplicated elements (i.e. UCO1, UAS1, UAS2, UGE1, UCL1, UCL2, UAT1, UOP1 and UPA1) within a scope (redundant type defect) is determined to be faulty without requiring model execution (i.e. testing is not required). Therefore, these operators are not considering in this work.

- A closer inspection of equivalent mutants generated by the WOP2 mutation operator (changes the visibility property of an operation) suggests that this operator generates an equivalent mutant when it is applied to a constructor operation because it only affects the access inherited by child classes (a private constructor of the super class is not inheritable). It is therefore impossible to detect this mutation operator when it is applied to a constructor operation. We therefore have to include this restriction in the rule of the WOP2 mutation operator to avoid generating this type of mutant.
- Changing a navigable association to a shared aggregation or vice versa (WAS2) generates an equivalent mutant because "aggregation=shared" has no semantic effect in a executable model using Alf [10]. Therefore, we only applied this operator changing from aggregation ="none" to aggregation="composite" or vice versa.
- Changing an Association Class to a Class with two associations or vice versa (WCL2 and WCL3). The association class effect can be equivalently modelled when the CS is expressed in Alf [11] (i.e. our execution environment).

The following operators could generate both and equivalent and non-valid mutants:

- Changing the visibility kind of an attribute (WAT4) generates both equivalent and invalid mutants, depending on whether the attribute is accessed internally by any member of the class (it is equivalent because everyone has access) or externally for any constraint that refers to this attribute through an association. In the last case, the mutant is non-valid and is detected by the parser.
- Changing a class abstract or vice versa (WCL4) when it does not result in a fault that the parser will detect when it tries to instantiate the class.
- Adding extraneous elements to CS (i.e. UCO2, UAS3, UAS4, UGE2, UCL3, UCL4, UAT2 and UOP2) generate equivalent mutants. Apparently, these operators did not inject a fault into a CS due to the nature of the test suite: only expected elements are tested. So, any additional element will remain untested. However, the operator that adds a Parameter to an Operation (UPA2) has to be considered because this affects a CS element (operation) that is tested by the test suite and so can be killed. These operators require a structural coverage analysis to be detected.

Finally, the operator that changes the order of the parameters in an operation (WOP1) generates a defect of inconsistency between the signatures of the CS operations and the operation calls from test cases. This defect affects the testing process more than the CS itself and also is detected by the parser. Therefore, this operator is not considered in this work. All the excluded operators generate mutants that require a static (without execution) technique for detecting.

Second iteration (Exclude High Order Mutants): We next analysed each derivation rule and identified the mutation operators that generate FOM and those that can generate HOM (see in Table 4 the relations between operators). Needless to say, if no other nested elements exist, this mutation operator also generates a FOM. For example, applying an operator to delete an operation (MOP) which has no parameters or related constraints generates a FOM. According to the CEH, the HOM are coupled

to simple mutants (FOM) in such a way that a test data set that detects all FOM will detect a high percentage of the HOM.

The operators that generate HOM are the following: WCO2, MOP, WCL2, WCL3, WCL4, WAT1, WAT2, WAT3, WOP3, MCL, MGE, MOP and MAT. We added restrictions to several of these operators in order to generate only FOM. Table 4 shows the 18 operators that we used in this work (marked with "*"), which were obtained as products of the described iterations. Figure 4 shows a partial view of a CS in which five mutation operators have been applied. Four operators will generate valid mutants and the MPA operator will generate a non-valid FOM because there is a class attribute (i.e. product_name) that is related with the parameter (p_atrproduct_name), therefore more changes (i.e. HOM) are required so as not to be detected by the parser. This CS is used in the literature to explain the development of a requirements model [19] which is used for our test case generation approach. This CS is included in our analysis in Section 5.

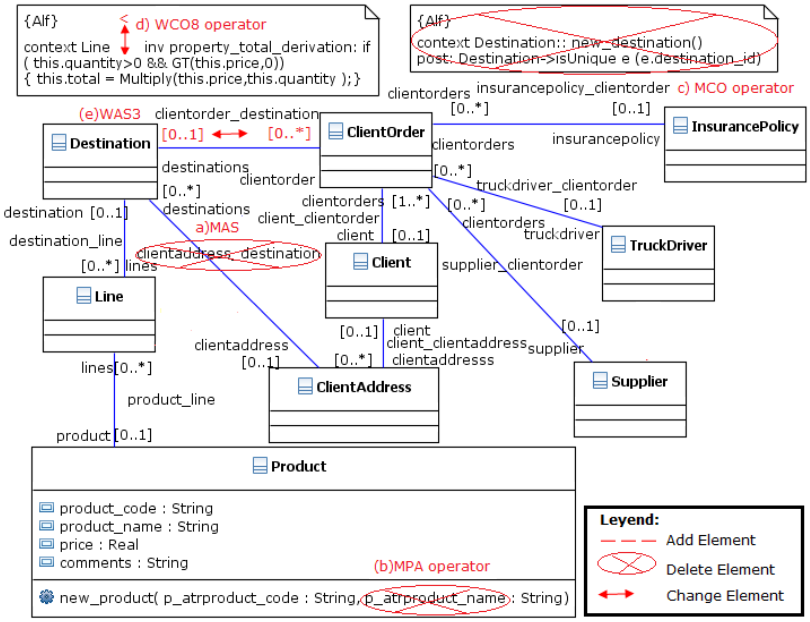

Fig. 4. Excerpt of a UML CD-based CS and the application of five mutation operators

5 Application and Analysis of Mutation Operators

The quality of mutants depends first on how well they reflect real errors that modellers make and second on whether they can be injected into a CS in such a way that they can be used for mutation testing. In order to analyse the effectiveness of the mutation operators, we used three conceptual schemas and respective test suites, which are described below.

5.1 Conceptual Schemas Under Test

We applied our mutation operators to three CS under test (CSUT) to evaluate the effectiveness of our mutation operators. These CS represent three kinds of systems: (i) the Super Stationery system (SS), which makes use of classes with attributes and

derived attributes, associations and constraints but has no generalizations, (ii) an Expense Report management system (ER) that uses fewer classes and relations but more constraints, and lastly, (iii) the Sudoku Game (SG) system [20], which is more variant-rich than the other two CS including generalization relations, derived associations and aggregations. The size of each CSUT is shown in Table 2 in terms of model elements.

Table 2. Elements of the Conceptual Schemas Under Test

Element	Super Stationery	Expense Report	Sudoku Game
Classes	9	7	11
Attributes	44	36	26
Derived Attributes	1	6	6
Operations	32	24	19
Parameters	91	75	48
Associations	9	8	6
Derived Associations	0	0	2
Composite Aggregations	0	0	3
Constraints	12	21	19
Inheritance	0	0	4

5.2 Mutant Generation

We developed a mutation tool prototype [21] to generate and analyse FOMs by applying the 18 selected mutation operators. This tool is divided into three distinct parts: a) calculate a mutants list, b) generate the mutants previously calculated; and, c) performing a syntactic analysis of the mutants. Figure 4 shows the number of valid and non-valid mutants generated by each mutation operator and CSUT.

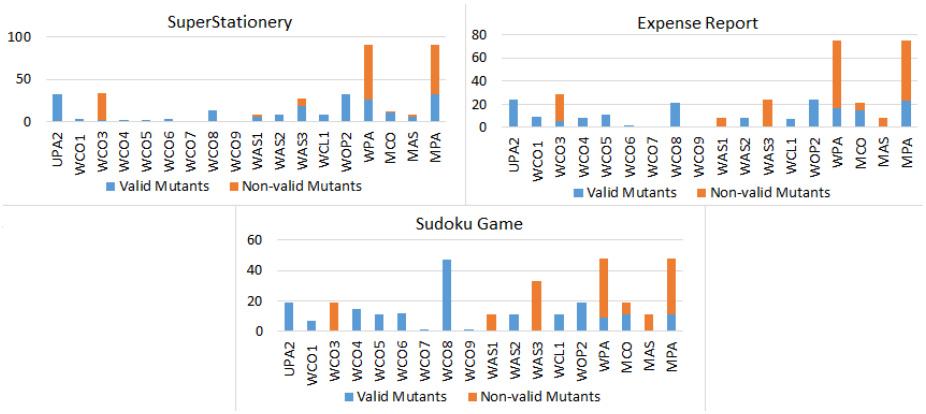

Fig. 5. Valid and non-valid mutants by each mutation operator.

The number of valid mutants produced by the WCO8 is the highest of the three CS (13, 21 and 47 respectively). Operators like UPA2, WCO1, WCO4, WCO5, WCO6, WCO7, WCO8, WCO9, WAS2, WCL1 and WOP2 generated only valid mutants for the CSUTs. However, the WCO7 and WCO9 operators generated only 1 mutant for the CSUT of the Sudoku Game system, giving a total 528 valid mutants (195, 159 and 174 respectively) and 495 non-valid mutants (171, 174, 150 respectively).

5.3 Mutation Testing Results

In this section, we assess the usefulness of the mutation operator for injecting faults in three CSUT. Test suites used in this study include tests checking all the CS class operations and constraints. Finally the data resulting from applying mutation testing to the CS were collected by applying the following measures.

For a conceptual schema CS and test suite T, M_T let the total number of non-equivalent mutants generated for CS and M_K (T) be the number of mutants killed by T. Mutation score for a test suite (MS (T) = M_K (T)/M_T) is the main measure used in mutation to measure the test suite effectiveness to kill mutants generated by applying all mutation operators. Where, non-equivalent mutants (M_T) = killed mutants (M_K) + the surviving mutants. The following measures, reflecting basic characteristics of mutation operators, were defined to evaluate the usefulness of the mutation operators [18]. Table 3 summarizes results of these calculations.

- Contribution Factor of mutation operator MO (CF (MO) = M_T (MO)/M_T). It shows to what extend mutants generated by applying mutation operator MO contributes to the total number of mutants generated for CS.
- Mutation Score of a mutation operator MO (MS (MO, T) = M_K (MO, T)/M_T (MO)). It shows the degree of detection for mutants generated by applying MO.
- Impact Indicator of a mutation operator MO (II (MO, T) = MS (T)-((M_K (T)-M_K (MO, T)) / (M_T-M_T (MO))). It shows how the mutation score obtained for T changes when operator MO was not applied.

Table 3. Results of mutation operator evaluation

CS / OP	Super Stationery			Expense Report			Sudoku Game		
	CF	**MS**	**II**	**CF**	**MS**	**II**	**CF**	**MS**	**II**
UPA2	0.16	1.00	0.080	0.14	1.00	0.064	0.10	1.00	0.119
WCO1	0.01	0.00	0.028	0.05	0.67	0.031	0.04	0.86	0.088
WCO3	0.01	0.50	0.037	0.03	0.80	0.040			
WCO4	0.01	1.00	0.042	0.05	0.75	0.036	0.07	0.54	0.063
WCO5	0.01	1.00	0.042	0.06	0.73	0.033	0.06	0.55	0.073
WCO6	0.01	1.00	0.038	0.01	1.00	0.043	0.06	0.36	0.057
WCO7							0.01	1.00	0.082
WCO8	0.06	0.69	0.034	0.11	1.00	0.054	0.22	0.68	0.058
WCO9							0.01	1.00	0.082
WAS1	0.03	1.00	0.046						
WAS2	0.04	0.00	0.004	0.05	0.0	0.000	0.06	0.00	0.038
WAS3	0.09	0.00	-0.035						
WCL1	0.04	1.00	0.050	0.04	1.00	0.047	0.06	1.00	0.101
WOP2	0.11	1.00	0.028	0.10	1.00	0.018	0.04	1.00	0.053
WPA	0.13	1.00	0.071	0.10	1.00	0.056	0.05	1.00	0.097
MCO	0.05	1.00	0.052	0.09	1.00	0.055	0.06	1.00	0.101
MAS	0.03	1.00	0.046						
MPA	0.16	1.00	0.080	0.13	1.00	0.063	0.06	1.00	0.101

For the SS, we ran 62 test cases. These test cases were executed against 206 mutated CS created by the mutation operators, killing 82% of the mutants. In the case of the ER, we executed 88 test cases against 174 mutants created, killing 90% of the mutants. For the case of the SG, we executed 90 test cases against 185 mutants, killing 74%. Therefore 89% of the mutation operators (16/18 operators) generate mutants that can detected by the test suites. More detailed information on the

mutation results can be found at https://staq.dsic.upv.es/webstaq/mutuml/mutation_operators.htm.

5.4 Discussion

The results in Table 3 show that the behaviour of the mutation operators may depend on some characteristics of the CS they are applied to (such as complexity of constraints, the number and type of elements included in the CS). However, the results suggest that some of these operators UPA2, WCO7, WCO9, WAS1, WCL1, WOP2, WPA, MCO, MAS, and MPA generated mutants that were relatively easy to detect by the provided test suites (the test suites had mutation scores of 100%). Moreover, all the operators had a "positive" impact (column value II>0) in the test suite assessment results. This means that the test suite quality is overestimated when any of these operators is not used. An underestimation of test quality, especially when the test suite is under development, would force an improvement of the test suite, while its overestimation could compromise the quality of any testing performed by them.

The mutation operators WCO1, WCO3, WCO4, WCO5, WCO6, WCO8, WAS2 and WAS3 all having a low mutation score, should always be applied because they generate hard to detect mutants and their application would stimulate selection of high quality tests. WAS2 and WAS3 mutation operators suggest that there is a lack of use (test) in the test suite of the CS elements affected by these operators.

Despite the mutation operator restrictions, all these mutation operators generated mutants in one or other of the three CS, these restrictions ensure that the mutants generated meet the condition "mutant has to be syntactically correct for mutation testing". Thus, these operators support a well-defined, fault-injecting process. Finally, mutation testing is computationally expensive, so it is important to use a technique that reduces the computational cost, the restrictions included in the mutation operator rules avoid generating non-valid mutants (495 in total in the three CS), which has practical benefits in the time saved in the mutation testing process. Additionally, the CEH states that complex (or higher-order) mutants are coupled to simple mutants (FOM) in such a way that a test data set that detects all FOM will detect a high percentage of the HOM.

6 Related work

Mutation Testing has been widely studied since it was first proposed in the 1970s by Hamlet [22] and DeMillo et al. [13]. In 2010, Jia and Harman [3] made a good survey of mutation techniques and also created a repository containing many interesting papers on mutation testing (last updated in 2014). This survey stated that mutation testing is mainly applied at the software implementation level (i.e. more than 50% of survey papers). But it has also been applied to models at the design level, for example to Finite State Machines [23], State Charts [24] and Activity Diagrams [25].

As far as we know, the idea of applying mutation testing to modify a UML CD-based CS and to assess the quality of test cases by checking how well they detect faults injected into a CS has not been explored to date in practice. However, some similarities can be found in Strug [26][18], Dinh-Trong et al. [17] and Derezinska

[16]. In the former [26], the author introduces nine mutation operators to apply manual mutations to the test suite provided for a UML/OCL-based design model instead of modifying the model, which is a different approach to that used in the present paper. In the latter [18], the author presents a classification of 16 mutation operators defined for constraints specified in OCL and used in UML/OCL-based design models. Constraints are among the CS elements covered by our approach. Dinh-Trong et al. [17] describe a set of mutation operators for a UML class diagram but do not include the restriction on generating valid mutants, neither do they investigate their usefulness in evaluating testing approaches. Finally, Derezinska introduced a set of mutation operators which can be applied to the UML CD specification but which are evaluated at the code level (C++) [16].

The present work is based on UML-based model defects classified in a previous work [9]. We also adapted some mutation operators proposed by Derezinska [16], Dinh-Trong et al. [17] and some operators for OCL constraints proposed by Strug [18]. Finally, in our approach the faults introduced include restrictions on generating only valid mutants for detecting in the CS at the analysis and design phases. This differs from current conventional mutation, in which the faults are introduced and detected at the code level.

7 Conclusions and Future Work

Mutation testing applied at the CS level can improve early development of high quality test suites (e.g. elements coverage) and can contribute to developing high quality systems (i.e. it meets requirements) especially in a model-driven context. In this paper we describe a mutation-testing based approach for UML CD-based CS level and report our recent work: (1) classifying a set initial of 50 mutation operators in the context of Conceptual Schemas based on a UML class diagram; (2) selecting and applying 18 mutation operators for FOM to evaluate the usefulness of the mutation operators in three CS. The main potential advantage of the defined mutation operators is that can support a well-defined, fault-injection process.

As opposed to code-based mutation, our mutation operators are based on the element characteristics of a UML CD–based CS and although some of the proposed operators perform syntactic changes at the constraints level, they are mainly focused (i.e. 41 of 50 operators) on the semantic changes of the high-level CD constructs. Our mutation operators are classified according to the element affected by the operator, injected defect type, and the action required by the mutation operator to generate valid mutants (syntactically correct). Since our purpose is to select mutation operators to be used to evaluate testing approaches, the selection process of mutation operators was divided into two iterations. In the first iteration, some operators were excluded because they generated only equivalent mutants (e.g. UCO2, UAS3, UAS4) and non-valid mutants, (e.g. WCL4, UCO1, UAS1), which require a static technique (without CS execution) for detecting (e.g. syntax analysis or structural coverage analysis), and so are not useful for mutation testing. In the second iteration, we aimed to analyse the dependencies between different operators and to reduce the cost of applying mutation testing by selecting 18 mutation operators that generate only first order mutants.

Based on the results obtained by applying the mutation testing, 56% (10/18) of our mutant operators generated a high number of killed mutants (score mutation=100 %). These results suggest that these operators generated mutants that are relatively easy to detect by the provided test suites. In the other case 44% (8/18) of the operators related to characteristics of associations (i.e. multiplicity and aggregation type) and constraints generated hard to detect mutants and their application would stimulate selection of high quality tests. However, the behaviour of the mutation operators may depend on the characteristics of the CS they are applied to, such as the number, element type and complexity of constraints.

This study is a part of a more extensive research project, whose principal goal is to propose an approach for testing-based conceptual schema validation in a Model-Driven Environment [3] [4]. Future work will proceed to extend the test suite for stimulating the disabled behaviour detected in this mutation analysis. We hope to evaluate the use of HOMs and compare them with FOMs. Finally, the proposed mutation analysis will be performed on a significant number of CS.

Acknowledgment

This work has been developed with the financial support of the Secretary of Higher Education, Science and Technology (SENESCYT: Secretaría Nacional de Educación Superior, Ciencia y Tecnología), of the Republic of Ecuador, European Commission (CaaS project) and Generalitat Valenciana (PROMETEOII/2014/039).

Appendix

Table 4. Mutation Operators defined for a UML CD-based CS

#	Code	Mutation Operator rule and relation with other mutation operators
1	UCO1	Adds a redundant constraint to the CD
2	UCO2	Adds an extraneous constraint to the CD
3	UAS1	Adds a redundant association to the CD
4	UAS2	Adds a redundant derived association to the CD
		UCO2
5	UAS3	Adds an extraneous association to the CD
6	UAS4	Adds an extraneous derived association to the CD
		UCO2
7	UGE1	Adds a redundant generalization to the CD
8	UGE2	Adds an extraneous generalization to the CD
9	UCL1	Adds a redundant class to the CD
10	UCL2	Adds an extraneous class to the CD
11	UCL3	Adds a redundant association class to the CD
12	UCL4	Adds an extraneous association class to the CD
13	UAT1	Adds a redundant attribute to a Class
14	UAT2	Adds an extraneous attribute to a Class
15	UOP1	Adds a redundant operation to a Class
16	UOP2	Adds an extraneous operation to a Class
17	UPA1	Adds a redundant parameter to an Operation
18	UPA2*	Adds an extraneous Parameter to an Operation
19	WCO1*	Changes the constraint by deleting the references to a class Attribute
20	WCO2	Changes the property (attribute) data type in the constraint
		WPA, WAT3
21	WCO3*	Change the constraint by deleting the calls to specific operation.
22	WCO4*	Changes an arithmetic operator for another and supports binary operators: +, -,*,/

23	WCO5*	Changes the constraint by adding the conditional operator “not”
24	WCO6*	Changes a conditional operator by another and supports operators: or, and
25	WCO7*	Changes the constraint by deleting the conditional operator “not”
26	WCO8*	Changes a relational operator by another and supports operators: <, <=, >, >=, ==, !=
27	WCO9*	Changes a constraint by deleting a unary arithmetic operator (-).
28	WAS1*	Interchange the members (memberEnd) of an Association.
29	WAS2*	Changes the association type (i.e. normal, composite).
30	WAS3*	Changes the memberEnd multiplicity of an Association (i.e. *-*, 0..1-0..1, *-0..1)
31	WGE	Changes the Generalization member ends
		MPA, UPA
32	WCL1*	Changes visibility kind of the Class (i.e. private)
33	WCL2	Changes Class by an Association Class
34	WCL3	Changes Association Class for a Class
35	WCL4	Changes the Class feature “isAbstract “ to true.
36	WAT1	Changes the Attribute feature “Is Derived” to true.
		UCO2
37	WAT2	Changes the Attribute property “Is Derived” to false
		MCO: Deletes the Attribute Derivation Constraint
38	WAT3	Changes the Attribute data type.
		WPA, WCO2
39	WAT4	Changes the Attribute visibility property
40	WOP1	Changes the order of the parameters
41	WOP2*	Changes the visibility kind of an operation. **Restriction**: WOP2 has to be applied to operations that are not related with any constraints.
		MCO
42	WOP3	Changes the data type returned by operation.
		WAT3
43	WPA*	Changes the Parameter data type (i.e. String, Integer, Boolean, Date, Real). **Restriction**: WPA has to be applied to parameters that are not related with attributes in a constructor operation. To reduce mutants only a change is counted.
44	MCO*	Deletes a constraint (i.e. pre-condition, post-condition constraint, body constraint)
45	MAS*	Deletes an Association. **Restriction**: MAS has to be applied to associations that are not related with any constraints.
		MCO
46	MGE	Deletes a Generalization relation
		MPA, UPA
47	MCL	Deletes the class (i.e. normal or association class)
		MCO: Deletes the constraints (body constraints) associated to the class
		MAT: Deletes the Class Attributes
		MOP: Deletes the Class Operations
		MGE: Deletes the Generalization relations related with the Class
48	MAT	Deletes an Attribute.
		MPA, MCO
49	MOP	Deletes the operation
		MPA, MCO, WCO3
50	MPA*	Deletes a Parameter from an Operation. **Restriction**: This mutation operator has to be applied to operations without related constraints.
		MCO

References

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

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

[3] M. F. Granda, “Testing-Based Conceptual Schema Validation in a Model- Driven

Environment," in *CAiSE 2013 Doctoral Consortium*, 2013.
[4] M. F. Granda, N. Condori-Fernandez, T. E. J. Vos, and O. Pastor, "Towards the automated generation of abstract test cases from requirements models," in *1st International Workshop on Requirements Engineering and Testing*, 2014, pp. 39–46.
[5] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *Softw. Eng. IEEE Trans.*, vol. 37, no. 5, pp. 1–31, 2011.
[6] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, 2009.
[7] Object Management Group, "Unified Modeling Language (UML)," 2015.
[8] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005, pp. 402–411.
[9] M. F. Granda, N. Condori-fernández, T. E. J. Vos, and O. Pastor, "What do we know about the Defect Types detected in Conceptual Models ?," in *IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, 2015, pp. 96–107.
[10] Object Management Group, "Action Language for Foundational UML (ALF)," 2013.
[11] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML)," 2012.
[12] IEEE, "IEEE Standard Classification for Software Anomalies," 2010.
[13] R. DeMillo, R. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer (Long. Beach. Calif).*, vol. 11, pp. 34–41, 1978.
[14] J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, 1992.
[15] M. R. Woodward, "Errors in algebraic specifications and an experimental mutation testing tool," *Softw. Eng. J.*, 1993.
[16] A. Derezińska, "Object-oriented mutation to assess the quality of tests," *Conf. Proc. EUROMICRO*, pp. 417–420, 2003.
[17] T. Dinh-Trong, S. Ghosh, and R. France, "A Taxonomy of Faults for UML Designs," in *In 2nd MoDeVa workshop - in conjunction with MoDELS*, 2005.
[18] J. Strug, "Classification of Mutation Operators Applied to Design Models," *Adv. Des. Manuf. V*, vol. 572, pp. 539–542, 2014.
[19] S. España, A. González, Ó. Pastor, and M. Ruiz, "Technical Report Communication Analysis and the OO-Method : Manual Derivation of the Conceptual Model the SuperStationery Co. Lab Demo," Valencia, 2011.
[20] A. Tort and A. Olivé, "Case Study: Conceptual Modeling of Basic Sudoku," 2006. [Online]. Available: http://guifre.lsi.upc.edu/Sudoku.pdf.
[21] "MutUML Tool," 2016. [Online]. Available: https://staq.dsic.upv.es/webstaq/mutuml.html.
[22] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279 – 290, 1977.
[23] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro, "Mutation Analysis Testing for Finite State Machines," in *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, 1994, pp. 220–229.
[24] S. Ferraz, J. C. Maldonado, T. Sugeta, and P. Masiero, "Mutation Testing Applied to Validate Specifications Based on Statecharts," in *Software Reliability Engineering, Proceedings. 10th International Symposium on*, 1999, pp. 210–219.
[25] U. Farooq and C. P. Lam, "Mutation Analysis for the Evaluation of AD Models," *Int. Conf. Comput. Intell. Model. Control Autom. CIMCA*, pp. 296–301, 2008.
[26] J. Strug, "Mutation Testing Approach to Evaluation of Design Models," *Adv. Des. Manuf. V*, vol. 572, pp. 543–546, 2014.
[27] I. Sommerville, *Software Engineering*, 9th edn. Boston: Addison-Wesley, 2011.