Towards a Functional Requirements Prioritization with early Mutation Testing

Nelly Condori-Fernandez
Department of Computer Science
University of A Coruña, Spain
Vrije Universiteit Amsterdam,
The Netherlands
n.condori.fernandez@udc.es
n.condori-fernandez@vu.nl

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

Tanja E. J. Vos
PROS Research Center
Universidad Politécnica de Valencia, Spain
Open University, The Netherlands
tvos@pros.upv.es

ABSTRACT

Researchers have proposed a number of prioritization techniques to help decision makers select an optimal combination of (non-) functional requirements to implement. However, most of them are defined based on an ordinal or nominal scale, which are not reliable because they are limited to simple operations of ranked or ordered requirements. We argue that the importance of certain requirements could be determined by their criticality level, which can be assessed using a ratio scale. The main contribution of the paper is the new strategy proposed for prioritizing functional requirements, using early mutation testing and dependency analysis.

KEYWORDS

Requirements prioritization; Mutation score; UML; dependency.

1. Introduction

In software industry, decision-makers often face the challenge of having more requirements than are possible to implement given different constraints, such as time, cost, and other scarce resources. Since it is crucial to distinguish the important requirements from the less important ones to maximize the overall business value [1], researchers have proposed a number of prioritization techniques to help managers, architects, designers select an optimal combination of (non-) functional requirements to implement. However, according to previous systematic reviews ([1], [3-7]), most of the existing requirements prioritization techniques are not scalable and reliable.

One of the reasons is that most of the criteria that must be taken into consideration when prioritizing requirements are very subjective. And most of these techniques used nominal and ordinal scale.

As traceability between requirements and system test cases is highly required, in the last years several approaches have been successfully proposed for generating automatic or semi-automatic test scenarios from requirements specifications (e.g. [11], [14]). In this paper we introduce the idea of applying mutation testing in functional requirements prioritization. Mutation testing is a well-known technique of assessing the quality of test suites.

In the context of requirements prioritization, we determine the importance of each functional requirement based on its criticality level. Such criticality is determined by test scenarios that contain more faulted test cases and dependencies with other test cases.

We consider that using an early mutation testing approach [2] (i.e. at model level) within our prioritization process will contribute to enhance the reliability of prioritization results.

The structure of the paper is organised as follows: Section 2 discusses the related work on requirements prioritization. Section 3 introduces the terminology on mutation testing and list the tools that we developed in the context of model-driven testing. In Section 4, we present the new functional requirements prioritization strategy, which is illustrated with an example (Sudoku game). Section 5 concludes the paper and suggests future work.

2. Related Work

Several secondary studies have been reported (e.g. [1], [3-7]) with the purposes of getting a better understanding on certain aspects of requirements prioritization techniques. For instance, Achimugu et al. [3] analysed 49 requirements prioritization techniques with respect to scalability, computational complexity, rank updates, requirements dependencies, communication among stakeholders, reliability, and validation. Sher et al. conducted another systematic review [4], who extended the analysis carried out by Achimigu et al. by considering not only technical aspects but also business/client aspects (i.e. sales, marketing, customer satisfaction, strategic) as analysis criteria of 59 prioritization techniques. From these two secondary studies, authors conclude that most of the existing techniques are not scalable. This was also corroborated by Khan [5], who found that most of the selected requirements prioritization techniques (i.e. AHP, Cumulative voting, Hierarchy AHP, Spanning Tree, Bubble Sort, Binary Search Tree, Priority Groups) are evaluated with a low number of requirements (an average of 18 requirements). This scalability problem of referenced rank techniques like AHP, Pairwise comparisons and Bubble Sort is because of the requirements are compared based on possible pairs.

Another issue is with respect to the reliability of the prioritization techniques. According to [3], the analytical hierarchy process (AHP) is one of the few techniques that provides reliable prioritization results because of its ability of determining ratios between requirements. Such ratios can be utilized in algorithms that can support the prioritization process.

Most of the existing techniques are not reliable enough since they use ordinal or nominal scale. A nominal-based prioritization technique (e.g. Top ten) has the capacity of showing the ranks of various requirements but cannot further indicate the extent at which, one requirement is considered to be more important than the other. Whereas ordinal-based prioritization techniques (e.g. Game planning, Cumulative voting, Ranking) are not able to quantify the relative priority difference among the ordered set of requirements. Herrmann and Daneva [6] conducted also a comparative review of 15 prioritization techniques with particular focus on cost-value techniques. In another mapping study [7], Pergher and Rossi found a limited research on the evaluation of tools and frameworks for requirements prioritization.

In this paper, we aim to address some of the issues detected in these secondary studies like reliability and scalability.

3. Background

This section introduces testing and mutation concepts used along the paper.

A. Testing Concepts

A **System Under Test** is a Conceptual Schema (CS) under test (CSUT) based on Unified Modelling Language (UML) Class Diagrams (CD) [8].

A **test suite** is a set of one or more test scenarios. Each test scenario is a story that consists of one or more test cases that execute in an incremental way the test scenario. A **test case** starts with the execution of the fixture. The fixture for a test case, at model level, is a set of statements that create a model state defining the values of the model variables. Then, the test case ends with an assertion that validates a functional requirement in the artefact under test (e.g. CSUT). A **Faulted test case** is obtained when the output of its execution is different than expected.

B. Mutation Concepts

Mutation involves modifying a software artefact (e.g. CS) by injecting artificial faults. Each mutated version is called a mutant. The artificial faults can be created automatically, using a set of mutation operators (MO). Mutants can be classified into two types: First Order Mutants (FOM) and Higher Order Mutants (HOM). FOMs are generated by applying mutation operators only once. HOMs are generated by applying mutation operators more than once [9].

In a previous paper [10], we proposed a set of 50 mutation operators specifically designed to generate mutants for UML Class Diagram. A subset of mutant types were evaluated respect its effectiveness to inject defects into a CS.

With the purpose of determining the ability of a test suite to expose errors in the system under test (i.e. CS), **Test suite adequacy** for each mutant type can be measured by a *mutation score* (MS). It is computed in terms of the ratio of the number of killed mutants (M_k) over the total number of the non-equivalent mutants. Where, non-equivalent mutants (M_T) = killed mutants (M_K) + the surviving mutants. The mutation score is given between zero and one values.

In the context of model-driven testing, we developed two tools:

1) the MμtUML tool (Mutation for UML) [2] for the generation and parsing (i.e. syntax analysis) of first order mutants by using the set of 18 mutation operators (see Table 1) previously defined for Conceptual Schema based on UML Class Diagram (CD); and,

2) the CoSTest tool [11] developed for testing conceptual schemas. CoSTest allows the semi-automatic generation of test scenarios with test cases from a requirements model, the execution of CS/CS mutants against generated tests, and reporting the test results. MμtUML has been integrated into CoSTest tool for supporting the test cases selection. Moreover, with CoSTest traceability between test cases and requirements is addressed.

4. New Requirements Prioritization

In this section, we explain our functional requirements prioritization strategy which consists of seven steps (See Figure 1). It is illustrated with an example of the Sudoku game. A brief description of this example is given in the Appendix section.

1) ***Evaluation of the test suite adequacy***. As it is shown in Figure 1 the output of the MμtUML tool [2] is the input of our prioritization strategy. Test suite adequacy evaluation is performed, where we compared automatically 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. A mutant may survive (S) either because it is equivalent (M_E) 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. The obtained mutations scores are the outcome of this step.

Table 1. Mutation Operators for First Order Mutants taken from [10]

#	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.

2) ***Ranking of the test suites***. As our purpose is identifying the critical test scenarios, in this second step, the values of the test suite adequacy (obtained in the previous step) are ranked in *ascending order*. Table 2 shows the mutants of our example with a mutation score lower than 1.0.

3) ***Adequacy score selection***. In the context of requirements prioritization, *a low adequacy score* should be selected because it will allow us to identify mutants type that are hard to kill and consequently also the requirements used to generate the corresponding test cases, which should be considered for a deeper review. In our example, we decided to select the first four mutants with MS lower than 0.6 (see

Table 2). The remaining 8 mutants had a score of 1.0, which were excluded. This step is carried out manually since required the selection of adequacy score to be used as a threshold for the next step.

4) ***Scenarios identification***. The (test) scenarios are identified by recognizing the faulted test case (i.e. expected verdict is fail and actual verdict is pass). The identified test scenarios correspond to sequence of events of the functional requirements model. In our example (see Figure 3), there is only one decision node, hence the cyclomatic complexity will be 2, which gives the number of independent paths or test scenarios (TS): TS1=Register Player, Sudoku Generation, Sudoku Cells, Put a Value in a Cell, Change to Unfinished Game, Sudoku Selection, Continue Playing; and TS2=Register Player, Sudoku Generation, Sudoku Cells, Put a Value in a Cell, Change to Unfinished Game, Sudoku Selection and End Game. Table 3 shows the test cases corresponding to the test scenario 1.

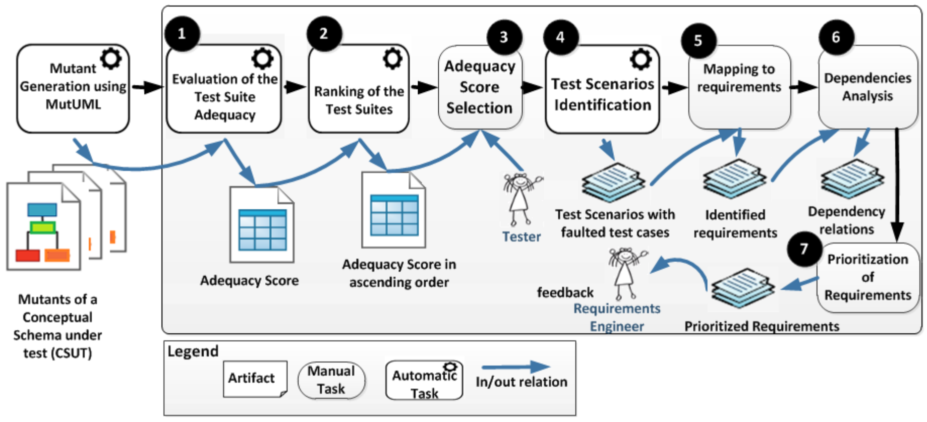

Mutant Generation using MutUML
1
Evaluation of the Test Suite Adequacy
2
Ranking of the Test Suites
3
Adequacy Score Selection
4
Test Scenarios Identification
5
Mapping to requirements
6
Dependencies Analysis
Mutants of a Conceptual Schema under test (CSUT)
Adequacy Score
Adequacy Score in ascending order
Tester
Test Scenarios with faulted test cases
Identified requirements
Dependency relations
7
Prioritization of Requirements
feedback
Requirements Engineer
Prioritized Requirements
Legend
Artifact
Manual Task
Automatic Task
In/out relation

Fig. 1. Functional Requirements Prioritization Strategy

Table 2. Mutation Score of test suites for each mutant type

CS Mutant type	Mutant Description	M_k	S	M_E	MS
WAS2	Changes the association type (i.e. normal, composite)	0	11	0	0.00
WCO6	Changes a conditional operator for another and supports operators: or, and	4	8	1	0.33
WCO4	Changes an arithmetic operator for another and supports binary operators: +, -,*,/	7	8	2	0.47
WCO5	Changes the constraint by adding the conditional operator “not”	6	5	0	0.55
WCO8	Changes a relational operator for another operators: <=, >=, ==, !=	28	13	6	0.68
WCO1	Changes the constraint by deleting the references to a class attribute	6	1	0	0.86

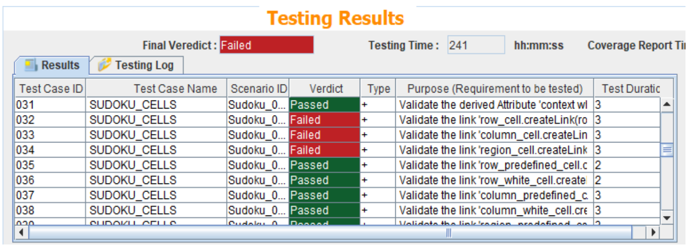
5) ***Mapping test cases to requirements.*** For each faulted test case, the related functional requirement is identified as it can be seen in Table 3. To do this, the CoSTest’s testing report is manually analysed. Figure 2 shows some of the testing results obtained for one mutant of the type *“Changes the association type*” (WAS2), where the test cases 32-34 were failed.

Table 3. Mapping test cases to functional requirements and number of dependencies

Functional Requirement	CS Mutant (Id Test cases)	Count
SUDOKU_CELLS	WAS2 (32, 33, 34, 41, 42), WCO6 (31, 41, 47), WCO5 (30, 31, 41, 47), WCO4 (41),	13
SUDOKU_GENERATION	WAS2 (19, 20, 21), WCO6 (8, 11, 14), WCO5 (8),	7
REGISTER_ PLAYER	WCO4 (4)	1
SUDOKU_SELECTION	WAS2 (55)	1
PUT_A_VALUE_IN_A_CELL	WAS2 (46)	1

6) ***Dependencies analysis***. Each test case has a dependency relation with the requirement that it validates. Then, the requirements are ordered according to the number of dependencies of failed test cases by mutant type (See last column of Table 3).

7) ***Prioritization of Requirements***. The requirement with the greatest number of dependencies and lower adequacy score represent the most critical requirement. The results in Table 3 shows that the highest level of criticality is for configuring number of cells for Sudoku (SUDOKU_CELLS requirement) with 13 dependencies, and the lowest level of criticality is for the requirements: REGISTER_PLAYER, SUDOKU_SELECTION and PUT_A_VALUE_IN_A_CELL. The SUDOKU_CELLS requirement is according to our example the most critical requirement because there are several defects injected in the mutants (i.e. WAS2, WCO6, WCO5 and WCO4) that affect this requirement and that they are not detected by test cases. From these results, we can see that requirements related with associations (WAS) and constraints (WCO) are critical and require more attention in their specification. These results were confirmed in the others six CSs reported in [12].


Testing Results

Final Veredict : Failed   Testing Time : 241 hh:mm:ss   Coverage Report Ti

Results | Testing Log

Test Case ID	Test Case Name	Scenario ID	Verdict	Type	Purpose (Requirement to be tested)	Test Duratic
031	SUDOKU_CELLS	Sudoku_0...	Passed	+	Validate the derived Attribute 'context wl	3
032	SUDOKU_CELLS	Sudoku_0...	Failed	+	Validate the link 'row_cell.createLink(ro	3
033	SUDOKU_CELLS	Sudoku_0...	Failed	+	Validate the link 'column_cell.createLin	3
034	SUDOKU_CELLS	Sudoku_0...	Failed	+	Validate the link 'region_cell.createLink	3
035	SUDOKU_CELLS	Sudoku_0...	Passed	+	Validate the link 'row_predefined_cell.c	2
036	SUDOKU_CELLS	Sudoku_0...	Passed	+	Validate the link 'row_white_cell.create	2
037	SUDOKU_CELLS	Sudoku_0...	Passed	+	Validate the link 'column_predefined_c	3
038	SUDOKU_CELLS	Sudoku_0...	Passed	+	Validate the link 'column_white_cell.cre	3

Fig. 2. A partial view of a testing results for a mutant of the type WAS2

Finally, the derived information allows requirements engineers to judge and weigh the requirements in a more objective way by computing the test suite adequacy and identifying the dependency value between the faulted test cases and the related requirements. This enables them to make decisions based on the criticality level of the requirements. This way decision makers (e.g. requirements engineer, project manager) can focus more on requirements that require a major attention due they are more fault-prone during their specification at model level.

5. Conclusions and Future Work

In this paper, we propose a new prioritization strategy that enables to select the most important functional requirements based on their criticality level. Our approach is promising because priorities are defined in a ratio scale (i.e. mutation score, number of dependencies), which has demonstrated to be more reliable than nominal/ordinal scale-based prioritization techniques. However, it is important to remark that the criticality level (criteria used for prioritizing requirements) could be influenced by the type of mutants that are generated with the mutation tool. Currently, we have used only first-order mutants. We think that adding high-order mutants would enhance the reliability of the prioritization results (i.e. more dependencies could be detected).

As future work, we plan conduct experiments for investigating the influence of this type of mutation operators on requirements prioritization. Moreover, we are going to automate the proposal (i.e. mapping and dependency analysis), and use software projects with a high number of requirements to assess the scalability of our approach .

References

[1] G. Ruhe, A. Eberlein, and D. Pfahl, "Quantitative WinWin — A new method for decision support in requirements negotiation," in *Proc. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 02)*, 2002, pp. 159–166.

[2] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and Ó. Pastor, "A Model-level Mutation Tool to Support the Assessment of the Test Case Quality," in *25TH International Conference on Information Systems Development (ISD2016 POLAND)*, 2016.

[3] P. Achimugu, A. Selamat, R. Ibrahim, M.N. Mahrin, "A systematic literature review of software requirements prioritization research," *Inf. Software. Technol.*, pp. 568–585, 2014.

[4] F. Sher, D. N. A. Jawawi, R. Mohamad, and M. I. Babar, "Requirements prioritization techniques and different aspects for prioritization a systematic literature review protocol," in *8th. Malaysian Software Engineering Conference (MySEC)*, 2014, pp. 31–36.

[5] K. A. Khan, "A Systematic Review of Software Requirements Prioritization," Blekinge Institute of Technology (BTH), Sweden, 2006.

[6] A. Herrmann and M. Daneva, "Requirements Prioritization Based on Benefit and Cost Prediction: An Agenda for Future Research," in *IEEE International Requirements Engineering*, 2008, pp. 125–134.

[7] M. Pergher, E. Massimiliano, and B. Rossi, "Requirements prioritization in software engineering: A systematic mapping study," in *Empirical Requirements Engineering (EmpiRE)*, 2013, pp. 40–44.

[8] Object Management Group, "Unified Modeling Language (UML)," 2015.

[9] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, 2009.

[10] M. F. Granda, N. Condori-Fernandez, T. E. J. Vos, and Ó. Pastor, "Mutation Operators for UML Class Diagrams," in *CAiSE 2016*, 2016.

[11] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor, "CoSTest : A tool for Validation of Requirements at Model Level," in *Requirements Engineering Poster and Demo*, 2017, pp. 4–7.

[12] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and Ó. Pastor, "Effectiveness Assessment of an Early Testing Technique using Model-Level Mutants," in *21st International Conference on Evaluation and Assessment in Software Engineering*, 2017, vol. 2017, no. June.

[13] A. Tort and A. Olivé, "Case Study: Conceptual Modeling of Basic Sudoku," 2006. [Online]. Available: http://guifre.lsi.upc.edu/Sudoku.pdf.

[14] C. Wang, F. Pastore and L. Briand, "System Testing of Timing Requirements Based on Use Cases and Timed Automata," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, 2017, pp. 299-309. doi: 10.1109/ICST.2017.34

Appendix

Figure 3 shows the sequence of functions of a software game called Sudoku, which was taken from Tort and Olivé work [13]. Sudoku system gives the capability of: managing different users (i.e. REGISTER PLAYER), generating new Sudokus (i.e. SUDOKU GENERATION), configuring number of cells for sudokus (i.e. SUDOKU CELLS). Then, the user puts value in a cell of the current Sudoku, the system checks whether the game ends to notify the user. The user should decide whether to continue playing by placing values in the sudoku cells (i.e. CONTINUE PLAYING and PUT A VALUE IN A CELL), ending the game or indicates to the system that he/she wants to continue solving an unfinished Sudoku previously started (i.e. CHANGE TO UNFINISHED GAME and SUDOKU SELECTION).

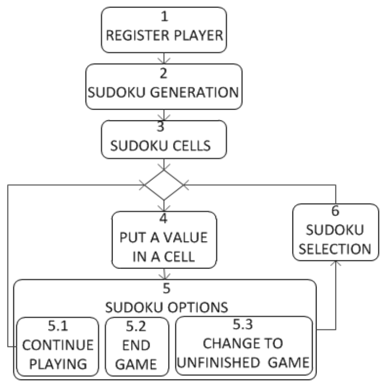


Fig. 3. Sequence of Events for Sudoku system