ModelDefenders: A novel gamified mutation testing game for model-driven engineering

Felix Cammaerts[1], Monique Snoeck[1]

[1]*Research Center for Information System Engineering, Leuven, KU, Belgium*

Abstract

Recently, there has been a surge in Model-Driven Engineering (MDE), where code is automatically generated from a model. While this has certainly enabled non-technical people to become something of a programmer, it doesn't necessarily make them good testers or good modelers. In mutation testing syntactic variations (mutants) are created from the source code and run against a test suite. Mutants that pass all the test cases in the suite are called alive, while mutants that fail are called dead. Good testers are able to develop test suites that kill all mutants. This can also be applied to MDE, where the mutants are created on the models used for code generation. This paper presents a gamified approach for mutation testing on models and discusses the specific challenges and caveats encountered when defining mutants and setting up such a gamified approach.

Keywords

Mutation testing, MDE, Education

1. Introduction

Mutation testing is a code-based testing technique in which syntactic deviations of the system under test (SUT) are generated, under the assumption that programmers write near-correct code. The mutants are run against the test suite used for the SUT. Mutants that pass all test cases are said to be alive, while mutants that fail one or more test cases are said to be dead. The ratio of live mutants gives an indication of how well the SUT has been tested. The living mutants can be used as feedback for the programmer, as they are different from the SUT, but have still managed to pass all the test cases, indicating a possible incomplete testing of the SUT or required changes to the code.

Mutation testing can also be applied to MDE, in two different ways. A first approach is to create mutants on the generated code to check whether the transformations work correctly [1]. A second approach is to create mutants on the models. Here, deviations in the modelling constructs of a model cause different outputs of the generated source code [2]. Since our focus is on improving the modelling skills of non-technical students, we will concentrate on the latter approach in this paper with the goal of proposing a gamified educational tool.

Companion Proceedings of the 16th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling and the 13th Enterprise Design and Engineering Working Conference, November 28 – December 1, 2023, Vienna, Austria

felix.cammaerts@kuleuven.be (F. Cammaerts); monique.snoeck@kuleuven.be (M. Snoeck)

https://www.kuleuven.be/wieiswie/nl/person/00143708 (F. Cammaerts);
https://www.kuleuven.be/wieiswie/nl/person/00012755 (M. Snoeck)

0000-0002-0037-3865 (F. Cammaerts); 0000-0002-3824-3214 (M. Snoeck)

This paper presents ModelDefenders, a gamified approach to introducing mutation testing in MDE. ModelDefenders is intended to be used as an educational tool where students learn to become better testers and modellers by evaluating and creating mutants on these models.

2. Related work

A similar gamified approach, as the one being introdued in this paper, has been developed for code-based mutation testing, this gamified approach is called CodeDefenders [3] and was the main source of inspiration for this paper. CodeDefenders uses the same game mechanics that we will use. CodeDefenders also aims to teach novice testers how to adequately test a software program. Research has shown that CodeDefenders is well received by students and has positive learning effects, with students performing steadily better [4]. It has also been found that the test suites and mutants developed within the game are stronger than those from automated tools [5].

For the implementation of ModelDefenders, the MERODE MDE approach was chosen [6]. The method is supported by a modelling tool that provides different levels of support for developing models [7] and a companion prototyper that allows students to experiment with their models [8] which includes a feature that provides students with feedback on their manual actions [9, 10]. The availability of a code generator makes a good basis for implementing a model defender game. MERODE is actively being taught in two modelling courses in two universities, which also provides opportunities for experimental evaluation.

This paper presents the dynamics of the ModelDefenders tool, specifically for the artifacts used in the MERODE MDE approach. It is explained how test cases and mutants can be defined for those artefacts, and how these developed test cases and mutants can be used to engage students in the practice of modelling and testing in a gamified approach. We attempt to formulate an answer to the following research questions:

RQ1. How can test cases be defined on artefacts used within the MERODE MDE approach (i.e. Finite state machines and class diagrams)?

RQ2. How can syntactic changes (mutants) be defined on artefacts within the MERODE MDE approach (i.e. Finite state machines and class diagrams)?

RQ3. How can these test cases and mutants provide a gamified approach to teach students the practice of software testing and modelling?

3. Defining test cases

In code-based testing approaches, test cases are usually written in the same programming language as the SUT. In MDE, the modelling language constructs are usually only used to develop the model, without the ability to define test cases using the same modelling constructs. Therefore, it is necessary to properly define how test cases should be defined for the different artefacts in MDE. The artefacts used in the MERODE MDE approach are a class diagram (CD) and statecharts that model the dynamic behaviour of the object types (OT). MERODE uses a subset of statecharts, namely finite state machines (FSM).

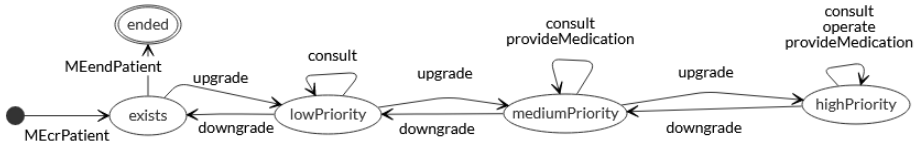

Figure 1: Finite state machine of the Patient model.

Figure 2: Class diagram of the Tuxedo model. The Tuxedo object type has as parameter `color` and the Person object type has as parameter `name`

3.1. Finite state machines

When defining a test case for an FSM, an execution sequence and an expected result should be provided. The execution sequence is a series of events that are executed sequentially on the FSM. The events in the test case should all be present in the FSM's alphabet. The result of the test case is a state in the FSM, or the error state if the execution sequence cannot be fully executed on the FSM. In the Patient model (Figure 1), the sequence of events *MEcrPatient*, *upgrade*, *upgrade*, *downgrade* would place the patient in the lowPriority state. While *MEcrPatient*, *upgrade*, *operate* is not possible as patient cannot be operated in the lowPriority state. The expected result is therefore an error state.

3.2. Class diagram

Similarly, when defining a test case on a CD, an execution sequence and an expected result should be provided. However, in a CD the sequence constraints are less explicit than in an FSM as there are no explicitly modelled states. Nevertheless, the sequence of instantiating and removing objects in the CD can be considered as an execution sequence. For example, an object p1 of Person must first be created before it can be removed. The expected outcome of such a sequence of events is either success or failure. Success if the entire sequence of events can be executed according to the multiplicities and relationships of the CD. Failure if one of the steps in the sequence is not possible at that point in the sequence, for example, deleting an object before it exists.

Specific to MERODE is that the relationships between objects express an existential dependency (ED) relationship, where one of the objects is the master and the other is the dependent. ED means that the master object must exist before the dependent object can be created and that the master object cannot be deleted until all the child objects have been deleted. In addition, a dependent can only be related to one master object throughout its life. For example, consider the Tuxedo model (Figure 2). Here Rental is dependent on Tuxedo and Person. This means that before a Rental can be initiated, there should already be a Tuxedo and a Person to which the Rental object can be associated.

When instantiating several objects of one OT, additional object pointers need to be provided to the events to identify the objects that are subject of the action. The minimum information needed

is the identification of the object that is impacted by the action, as well as the identification of related objects via object pointers when a new object is created. Object pointers are given between square brackets [], and paramerters between round brackets (). An execution sequence for a CD might look like this `crTuxedo[](Red): t1, crPerson[]('Felix'): p1, crRental[p1, t1]: r1`. This test case instantiates a Tuxedo t1, a Person p1 and a Rental r1 which is dependent on Tuxedo t1 and Person p1. The expected result of this execution sequence would be success. Conversely, the execution sequence `crTuxedo[](Red): t1, crRental[t1, p1]: r1, crPerson[]('Felix'): p1` would fail as the Person object must be created before the Rental object can be created. When taking into account the parameters of the object types as well, it is important to check whether the datatype of the given parameter matches with the data type of the MUT. If the parameter does not match, the test case is considered invalid.

4. Defining mutants

Mutation testing involves making small syntactic changes to the source code to create mutants, such as changing $<$ to $\leq$. These syntactic deviations are run against the test-set, and mutants that manage to pass all the test-cases in the test-set are an indication of bad or missing test-cases in the test-set. When defining such mutants in code-based approaches, it is important to note that these deviations usually keep the skeleton of the code the same (for example, using the same method calls, and classes keeping the same relationships to other classes). Similarly, when defining mutants for mutation testing of model-based approaches, it is important to clearly distinguish between what is considered to be the skeleton of the model and what parts can be mutated. This section therefore provides an overview of possible mutations.

4.1. Finite state machines

When creating a mutant for an FSM, the states of the MUT must remain unchanged. Mutations can be modelled by changing the labels of transitions and thus the events that would trigger those transitions. It is also possible to add and remove transitions. In addition, the following constraints should be observed to ensure that the test cases remain executable on the mutant: (1) The mutated FSM must not contain any nondeterminism, as this would make the outcome of the test case nondeterministic. (2) The mutated FSM must retain the names of the states as in the Model Under Test (MUT). Failure to do so would make it impossible to correctly verify the outcome of the test cases on the mutant.

4.2. Class diagram

When creating a mutant for a CD, the OTs of the MUT must remain unchanged. Mutations can be modelled by changing the multiplicities of the relationships between OTs. It is also possible to add and remove relationships and changing the datatype of the parameters. In addition, when modelling a mutant for a CD, the following constraints should be observed to ensure that the test case remains executable on the mutant: (1) No cyclic dependencies can be introduced in the

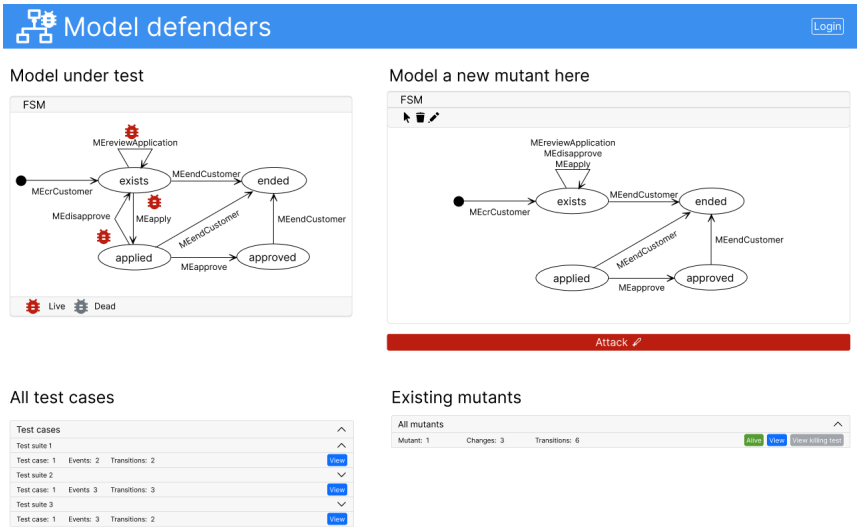

Figure 3: Attacker view when attacking an FSM as model under test.

mutant. (2) The mutated CD must retain the names of the OTs as modelled in the MUT. (3) The number of parameters and their positions must remain the same.

5. Gamification

To encourage students to develop mutants and test cases under the constraints mentioned above, a gamification approach can be used. This gamification approach is baptised *ModelDefenders*. In ModelDefenders, students are assigned one of two roles: attacker or defender. Defenders are tasked with creating test cases that consist of a sequence of events and an expected outcome (Section 3). The attacker is tasked with creating mutants on the models (CD and FSMs), according to the aforementioned rules (Section 4). Similar to mutation testing, the mutants (from the attacker) are run against the test cases (from the defender). A mutant that successfully passes all test cases is said to be alive, a mutant that fails at least one test case is said to be killed. If a mutant survives, the attacker gains one point, while if the defender's test cases have successfully killed a mutant, the defender gains one point.

5.1. Attackers

The attacker is given the MUT, which is either a CD or an FSM. Figure 3 shows an example of an FSM under test (top left). Below it is an overview of the test cases developed by the defender. The attacker can click on "View" to see the complete test case. This shows the full sequence of events from the defender and the expected outcome. At the top right, the attacker is given an editing area in which he can modify the MUT to model a mutant. Once the attacker is finished modelling the mutant, he can click on 'attack', which will run the current test cases against the mutant. If one of the test cases has a different outcome than the expected outcome for the

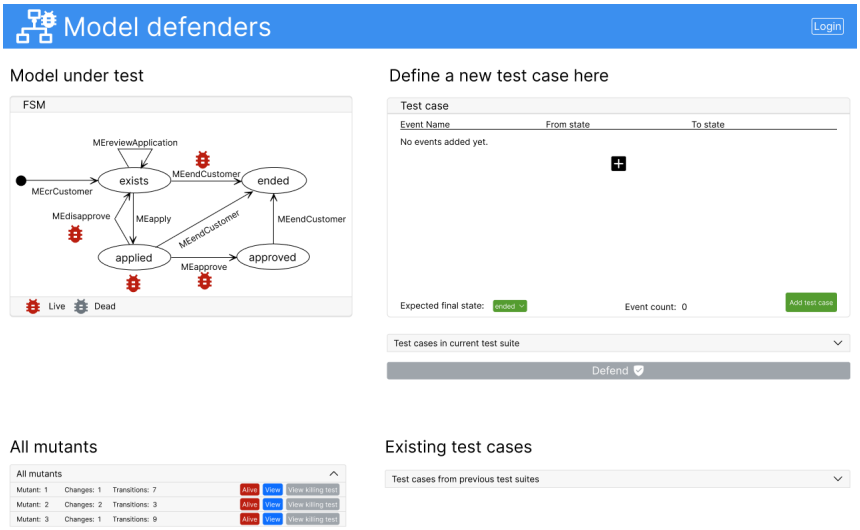

Figure 4: Defender view when defending an FSM as model under test.

mutant, the mutant is dead; if all the test cases have the expected outcome, the mutant is alive. On the bottom right the attacker is given an overview of the mutants that have previously been used as attack, including which are dead and which are alive. The attacker can click "View Killing Test" if the mutant is dead to see which of the tests the mutant failed. The attacker can click 'View' to see the mutant.

5.2. Defenders

The defender is also given the MUT, which is either a CD or an FSM, in the top-left corner (see Figure 4). The defender can define test cases for this FSM in the top right pane. Here, the defender can add each of the possible events of the FSM one by one and specify the expected outcome state (i.e. a state of the FSM or an error). Once a test case is fully defined, the defender can add it to the test suite. Before the test case is actually added to the test suite, it is checked whether the expected outcome of the test case actually matches the outcome that would be expected from the FSM. If this is not the case, the test case is not added to the suite and the user is informed that his test case has been incorrectly defined. Once the defender feels that he has fully developed the test suite, he can use it to defend against the mutants modelled by the attacker. These mutants are shown at the bottom left. Each mutant is labelled as dead or alive. The defender can look at the mutants to help define the test cases for the test suite. If a mutant is dead, the defender can also "view killing test" to see which of the test cases killed it.

5.3. Running test cases against a mutant

To check whether a mutant defined by an attacker is dead or alive, the defender's test cases are run against the mutant. If all the test cases have the same outcome on the mutant as on the

MUT, the mutant is alive. If at least one test case has a different outcome on the mutant than on the MUT, the mutant is dead.

For FSMs it is quite straightforward to check the outcome of the test case on the mutant. The sequence of events can be run on the mutant, the state of the mutant after executing the last event of the test case is the outcome of the test case on the mutant. This result can be compared with the result on the MUT. As soon as one of the events of the sequence cannot be executed on the current state of the mutant, the resulting state of the mutant becomes the error state.

Even though, adhering to the constraints imposed for defining mutants of a FSM would mean there is no explicit focus on common mistakes when modelling FSMs, such as liveliness aspects, these are still implicitly present. Namely, if the MUT is valid (i.e. contains no backwards/forward inaccessible states and this no liveliness problems), a well-defined suite of test cases, would be able to detect any mutant that does introduce these mistakes. For example, in Figure 1, a mutant that omits the upgrade transition between mediumPriority to highPriority, would be killed by the test case *MEcrPatient, upgrade, upgrade, upgrade* with as expected outcome highPriority. This should allow a defender to understand that the accessibility of each of the states should be tested. In this case this is done indirectly by defining a test case with an expected outcome state on the FSM. For an attacker this should allow to understand that the absence of such test cases allows for mutants to be designed which do not adhere to the liveliness properties of FSMs.

For CDs, it is not enough just to look at the order of the events; the parameters, object pointers and associations present in the MUT and the mutant should also be considered. For example, consider the Tuxedo case in figure 2. A tuxedo can be associated with 0 or 1 rentals, while a person can be associated with 0 or many rentals. Considering only the order of events, a test case `crTuxedo[](Red): t1, crTuxedo[](Blue): t2, crPerson[]('Felix'): p1, crRental[t1, p1]: r1, crRental(t2, p1): r1` with expected success, would be able to kill a mutant in which the relation between Tuxedo and Rental has been changed to a 0 to 1, since the last crRental is not possible on the mutant. However, this test case would not be able to detect a mutant in which the relationship between Tuxedo and Rental has been omitted. For this mutant the sequence of events `crTuxedo[](Red): t1, crTuxedo[](Blue): t2, crPerson[]('Felix'): p1, crRental[t1, p1]: r1, crRental[t2, p1]: r1` would also be successful. To take the object pointers into account, one should look at each association to objects present in the event of a test case. Take for example the event `crRental[t1, p1]: r1`. This test case creates two relations, one between t1 and r1 and another between p1 and r1. For each of the OTs that are part of these relations (Tuxedo-Rental and Person-Rental), the following rules should be checked. X is the master OT, Y is the dependent OT.

- If there is a direct relation between X and Y in both the MUT and the mutant, then check with the multiplicity of the mutant if a new object of type Y can be created.
- If there is a direct relation between X and Y in the MUT, but not in the mutant, no check for this specific relation is needed. However, the other relations still need to be checked.
- If there is no direct relation between X and Y in the MUT, but there is one in the mutant, check whether there already exists an object of type X at that point in the event sequence

and whether a new object of type Y can be instantiated on the mutant considering the multiplicity of the relationship.

Concerning parameters, it is sufficient to check whether at each object instantiation, the parameters used in the test cases are of the same data type as the parameter in the MUT at the same position. If the data types are the same, even though the values of the data might not match, then the mutant survives the test case. While if the data types are different, the test case kills the mutant.

We do acknowledge that most CDs are based on UML and do not restrict the relations into being existent-dependent relationships. Nonetheless, the way of defining mutants for CDs and defining test cases remains the same. The rules for comparing a test case against a mutant are now applicable to any X and Y which are directly related to each other.

6. Evaluation

ModelDefenders is currently under development. Mockups have been created in Figma[1] to understand the possible user interactions with the tool. These mockups are currently being translated into a web application. Once the development of ModelDefenders is complete, it can be evaluated. Firstly, the usability of the tool will be evaluated. Secondly, it will be assessed whether ModelDefenders motivates student to test more. Finally, we will assess whether this increased 'exploration' of models in turn increases students' understanding of modelling.

Acknowledgments

This paper is being funded by the ENACTEST Erasmus+ project number 101055874.

References

[1] A. Gonzalez, C. Luna, G. Bressan, Mutation testing for java based on model-driven development, in: 2018 XLIV Latin American Computer Conference (CLEI), IEEE, 2018, pp. 1–10.

[2] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, W. E. Wong, Model-based mutation testing—approach and case studies, Science of Computer Programming 120 (2016) 25–48.

[3] J. M. Rojas, G. Fraser, Teaching mutation testing using gamification, in: European Conference on Software Engineering Education (ECSEE), 2016.

[4] G. Fraser, A. Gambi, M. Kreis, J. M. Rojas, Gamifying a software testing course with code defenders, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, 2019, pp. 571–577.

[5] J. M. Rojas, T. D. White, B. S. Clegg, G. Fraser, Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 677–688.

[1] https://www.figma.com/file/lJ6HVBH0hmJGz2RSTMQMc6/ModelDefenders?type=design&node-id=0%3A1&mode=design&t=siCzSe9kKoYsK4oa-1

[6] M. Snoeck, Enterprise information systems engineering, The MERODE Approach (2014).
[7] M. Snoeck, MERLIN: An Intelligent Tool for Creating Domain Models, in: Research Challenges in Information Science: 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23–25, 2020, Proceedings 14, Springer, 2020, pp. 549–555.
[8] G. Sedrakyan, S. Poelmans, M. Snoeck, Assessing the influence of feedback-inclusive rapid prototyping on understanding the semantics of parallel UML statecharts by novice modellers, Information and Software Technology 82 (2017) 159–172.
[9] B. Marín, S. Alarcón, G. Giachetti, M. Snoeck, Tescav: An approach for learning model-based testing and coverage in practice, in: Research Challenges in Information Science: 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23–25, 2020, Proceedings 14, Springer, 2020, pp. 302–317.
[10] F. Cammaerts, C. Verbruggen, M. Snoeck, Investigating the effectiveness of model-based testing on testing skill acquisition, in: IFIP Working Conference on The Practice of Enterprise Modeling, Springer, 2022, pp. 3–17.