Software and Systems Modeling (2024) 23:953–971
https://doi.org/10.1007/s10270-023-01141-1

SPECIAL SECTION PAPER

Assessing the testing skills transfer of model-based testing on testing skill acquisition

Felix Cammaerts[1] · **Monique Snoeck**[1]

Received: 13 April 2023 / Revised: 10 October 2023 / Accepted: 29 November 2023 / Published online: 22 January 2024
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2024

Abstract
When creating a software model, it is necessary that it accurately captures the desired behaviour, while at the same time ensuring that any undesired behaviour is excluded. On the one hand, formal verification tools can be used to check the internal consistency of a software system, ensuring that the behaviour of one software component does not contradict another. On the other hand, software testing is essential to check the external validity of the model more comprehensively. Unfortunately, software testing is often overlooked in curricula, resulting in graduates with inadequate software testing skills for industry. Software testing tools such as TesCaV can be used to help teachers teach software testing topics in a non-intrusive and less time-consuming way. Previous research has shown that TesCaV is easy to use and that novice users produce better quality software tests when using TesCaV. However, it has remained unclear whether learners retain the skills they gain from using TesCaV even when the tool is not offered for help. In order to understand the positive effect of TesCaV on learners' software testing skills, this study conducted an experiment with 45 participants. The experiment used a pretest-treatment-posttest design. The results show that participants feel equally confident about the completeness of their test coverage, even though they identify more test cases. It is concluded that for course design, a capsule such as TesCaV can help students to understand the full complexity of software testing and help them to be more systematic in their approach.

Keywords Model-based testing · Model-driven engineering · TesCaV · MERODE

1 Introduction

Software testing is an important part of the software development lifecycle. Proper software testing ensures that deployed software systems perform as expected. This not only reduces the opportunity cost of customer churn, but also allows developers to focus on features for the next release, rather than fixing bugs.

In fact, a 2020 report estimated the total cost of poor software quality at $2.08 trillion in the US alone [14]. The report recommends that much emphasis be placed on preventing software defects. For example, early and regular analysis of source code to identify weaknesses and vulnerabilities, measuring structural quality characteristics, considering the weaknesses and vulnerabilities of embedded components, and adopting secure coding practices.

Software testing can be used to improve software quality. Software testing helps to avoid bugs, which allows developers to focus on features for a next release rather than fixing bugs. Software bugs have been found to account for 80% of the cost of a software system [23]. It has also been reported that developers spend 35–50% of their time validating and debugging software [1].

There is a lack of testing culture and awareness in academia [11, 25], resulting in little time being spent on software testing in education [33]. Instead, teachers place more emphasis on other topics such as requirements engineering and system design, leaving little time for software testing [7]. This has led to graduates failing to meet industry standards when it comes to software testing [18]: research shows that senior computer science students fail to provide adequate tests for even a small computer program [4]. In fact, software testers have been found to be inadequately trained in indus-

Communicated by Kurt Sandkuhl, Balbir Barn, Tony Clark, and Souvik Barat.

✉ Felix Cammaerts
felix.cammaerts@kuleuven.be

Monique Snoeck
monique.snoeck@kuleuven.be

1 LIRIS, KU Leuven, Naamsestraat 69, Leuven 3000, Flemish Brabant, Belgium

try standards for software testing and to have received most of their formal software testing training outside of university [5].

Although there has been an increased focus on software testing in education [10], software testing remains a complex and challenging subject to teach. Studies have vowed to introduce software testing early in the curriculum, ideally in an introductory programming course [11]. Free and open source software is often cited as beneficial to the learning process, allowing students to find bugs in code that is not their own, thus avoiding the negative setback of students finding bugs in their own code. Similarly, gamification approaches can further motivate students to gain experience in software testing [9]. However, teachers of such introductory programming courses are often overwhelmed by the larger number of topics they have to teach, leaving little time to teach software testing as a separate module in their classes.

The introduction of software testing tools has provided a possible solution to this problem. Software testing tools allow teachers to continue to focus on other topics while introducing their students to testing techniques in a less time-consuming way. The use of such software testing tools can also ensure that students are prepared to meet software testing industry standards. To achieve this, it is important that software testing is introduced taking into account the cognitive learning process of students [18].

An example of a software testing tool designed for educational purposes is TesCaV (TEst CoverAge Visualisation), which provides users with feedback on the manual tests they have performed on a software system [16]. TesCaV is part of MERODE, a model-driven engineering (MDE) approach in which a software system can be created based on a class diagram and statecharts. Users can generate a prototype application from the models and use it to simulate scenarios to test the models. During this process, TesCaV provides users with feedback on the adequacy of their tests, mainly in terms of coverage criteria.

Previous research has shown that for the subject of software testing, TesCaV is beneficial to the student learning process whilst also being user friendly [16]. In addition, the use of TesCaV has been found to result in students performing more comprehensive tests on a software system [3] when allowed to use the tool. However, no evidence has yet been found that TesCaV has a lasting positive effect on novice users' ability to achieve high test coverage, even when TesCaV is no longer used. Such a lasting positive effect is desirable, as it would provide evidence that novice users are able to transfer newly acquired testing skills from TesCaV to other scenarios. A longitudinal study with a control group would be needed to test for a truly long-lasting effect. While it might be possible to follow students over a semester, attributing improved performance to the treatment would require controlling for the many different factors at play in an educational context. The design of such an experiment is constrained by data protection and the obligation to treat all students equally. Henceforth, this research will focus on the short-term lasting effect and address the following research question: *Does providing TesCaV to novice users enable them to transfer the acquired knowledge of test coverage to new cases?* In order to answer this research question, a new experiment will be conducted. This experiment is designed to specifically measure the knowledge transfer effect after the treatment, in this case the use of TesCaV.

The rest of this paper is structured as follows. Section 2 discusses related work in terms of software testing tools, model-based testing, and the MERODE approach into which TesCaV has been integrated. Section 3 presents the experimental setup used for this research. Section 4 presents the results of the experiment, which are then discussed in Sect. 5, where internal and external validity are also discussed. Section 6 presents the conclusion of this research as well as further research possibilities.

2 Related work

2.1 Teaching of software testing

There are numerous software testing tools designed to support teaching, many of which use white-box testing to evaluate the internal implementation of code.

Web-CAT is an online tool designed to help students learn how to write effective code and test cases based on a given specification [8]. It uses the Test-Driven Development approach, which involves writing test cases for a software system before actually implementing it. To use Web-CAT, students are given a specification for a software program and asked to write source code that satisfies the specification, as well as test cases to ensure that the code works correctly. Once submitted, the tests are run against the code and students receive feedback on any incorrect or missing tests. Students can resubmit their code and tests as many times as necessary to improve their results. This feedback is crucial in reinforcing the value of Test-Driven Development, as it highlights the importance of continually creating new tests for each new implementation to ensure that the software works correctly.

Another approach to teaching software testing is the CodeDefenders framework [21], which uses mutation testing to help students understand proper testing practices. Mutation testing is a technique in which small changes are made to the original program to create "mutants", which are then tested against test cases. If a mutant fails a test case, it is considered "dead", whereas a mutant that passes all test cases is "alive" and can be used to create new mutants. The mutation score is the ratio of live mutants to the total number of mutants, which indicates how well the program has been tested. In the Code

Defenders game, students are divided into 'defenders' and 'attackers'. Defenders write test cases to kill mutants, while attackers write mutants to survive test cases. Defenders earn points for killing mutants, while attackers earn points for having surviving mutants. CodeDefenders has been shown to actively engage students and improve their testing skills [9].

Martinez [19] used a method called Question-Driven Teaching to help students learn how to write good test cases. This method involved presenting students with an exploratory test case along with guiding questions to analyse it. The resulting list of questions generated by the students was used by the teacher to explain the rationale behind the test strategy. According to the research, this approach led to improved test quality in real-world applications.

Testing Tutor is a web-based assignment submission platform with a customisable feedback engine that supports different levels of testing pedagogy [6]. A study evaluated different forms of feedback, with researchers concluding that students who received conceptual feedback achieved greater code coverage with fewer redundant test cases compared to those who received detailed feedback [6].

Marmoset is an automated submission and testing system that provides feedback to both teachers and students [29]. Lecturers use tokens to grant access to their private test cases, motivating students to start testing early and allowing lecturers to identify and address student difficulties. Students have reported positive experiences with Marmoset [29].

Finally, Sarkar and Bell [24] introduced a black-box testing tool for acceptance testing, which received feedback from final year students for potential improvements.

Although there has been some research into teaching software testing, most of these methods are aimed at people who already have technical expertise, as they focus on code-based methods, mainly white-box testing. They aim to improve the existing understanding of these methods rather than to teach beginners how to start testing properly. All of these strategies require students to manually find and execute all possible test cases. As a result, students may unintentionally run equivalent test cases multiple times, or be completely unaware of other test cases, resulting in incomplete program testing.

2.2 Model-based testing

Model-Based Testing (MBT) is a type of black-box testing method that can automatically generate test cases from a given software model. This method helps to address the challenge of manual testing by automating the test case generation process. The definition of a test case can be found in the ISTQB [12]: "A set of input values, execution preconditions, expected results and execution postconditions developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement". MBT involves a series of steps that have been defined by [31]. An overview of these steps is also given in Fig. 1.

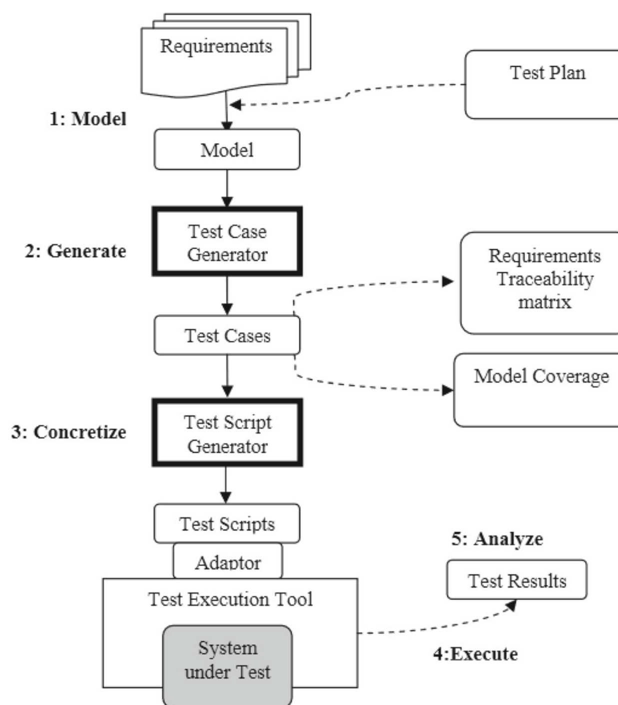


Fig. 1 Overview of the steps taken in an MBT approach [31]

1. Model: The system requirements are used to create a software model. A software model is an abstract representation of the system under test. This step is done manually. An example of such a test model would be a state machine or a set of state machines describing the behaviour of the system under test.
2. Generate: A set of abstract test cases is generated based on specific test selection criteria such as transition coverage. Such a specific test selection criterion is needed because the number of possible tests can be infinite. These criteria form part of the test plan. The test cases generated in this step are considered abstract because they are designed based on the software model defined in the first step and are not directly executable on the system under test. These abstract test cases can already be used to produce a model coverage report, i.e. how well the test selection criteria cover the model, and a requirement traceability matrix, which links individual test cases to functional requirements.
3. Concretise: The abstract test cases are transformed into concrete test cases. These concrete test cases are directly executable on the system under test. This step can be performed by a separate transformation tool. Such a two-step approach allows the independent generation of tests that

can be reused in other systems under test, regardless of their implementation language.

4. Execute: The concrete test cases generated in the previous step are executed. A report can be generated on the test case executions that have produced a result that differs from the expected result.
5. Analyse: Once the test cases have been executed, the results are analysed and any defects in the code are identified and fixed. The analysis of the test case execution results is done manually. Of particular interest is any test case execution that has produced a result different from the expected result.

Models created in a modelling language such as UML can be used as input for automatic test case generation [17]. The process of transforming abstract test cases into executable ones can be done either manually or automatically. In [17], a testing methodology has been introduced that automatically generates executable test cases from the abstract ones. This new approach significantly reduces the time needed for testing. The abstract test cases are generated using nine test criteria based on [30]. Pérez and Marín [20] have developed a software tool called TCGen that automatically generates abstract test cases from UML conceptual models. The generation of abstract test cases is based on a set of test criteria. Compared to manual generation of abstract test cases, TCGen has been shown to deliver better results in terms of efficiency and code coverage.

2.3 MERODE approach

The MERODE approach allows the creation of a conceptual model from which a fully functional Java application can be generated using the MERODE code generator, making it a Model-Driven Development approach. Users can start from a requirements text that specifies the requirements for the software system to be modelled. These requirements can be modelled using a UML class diagram containing the domain object types for the software system. For each domain object type, a Finite State Machine (FSM) can also be modelled, specifying the life cycle of that object type in terms of states and transitions. The combination of this UML class diagram and the associated FSMs forms the conceptual model for the software system. An example of such a conceptual model is given in Fig. 2.

However, there are several possible sources of error. Firstly, it is possible that the requirements from the requirements text have been incorrectly modelled. Secondly, the conceptual model may contain internal inconsistencies, such as deadlocks in the FSMs. Finally, there is no guarantee that the automatically generated code is a correct translation of the conceptual model. An overview of the MERODE approach and possible discrepancies is given in Fig. 3.

Once users have created a conceptual model for the software system, they can generate a Java application using the MERODE code generator. The generated Java code contains code according to the data and behaviour defined by the conceptual model, as well as additional code to obtain an executable application, which serves as a prototyper tool [26]. This prototyper allows to quickly validate whether the requirements are consistent with the conceptual model and thus with the generated Java code. In the MERODE approach, business events are the units of interaction between the environment and the information system. The user interface of the generated application allows to trigger the execution of business events (e.g. create a person, create a tuxedo, rent a tuxedo). The sequence of executed events and their results can be considered as test cases.

The prototyper application includes feedback related to the models used to create it. Previous studies have shown that providing such feedback-enriched prototyping is an effective approach for enhancing novice users' understanding of conceptual models and improving model quality [27]. However, as this feedback is only provided to the user when an illegal action is attempted (e.g. attempting to rent a tuxedo that is already on loan), this type of feedback can be considered as immediate informative negative feedback [28]. Illegal actions result from constraints in the model (e.g. a maximum multiplicity of one). The constraints may have been deliberately included by the modeller, or they may result from modelling errors. In the latter case, students are likely to discover their errors by testing scenarios that they expect the system to accept. However, detecting the correct modelling of deliberately included constraints requires the deliberate creation of scenarios to assess the correct modelling of constraints. Indeed, previous experiments have also highlighted the limited testing skills of students, as evidenced by their use of an insufficient number of test cases to assess the accuracy of requirement modelling [27].

Figure 4 gives an overview of the general functionality of the prototype application using the code generated from the conceptual model in Fig. 2. The application contains one tab for each domain object type. The tab displays the objects (instances) that have been created and a button for each of the actions (business events) that can be performed on these objects. The buttons are grouped according to the type of action: on the left are the creation events (these create a new instance of the object type), in the middle are the modification events (these change the state of an instantiated object), and on the right (not shown in the figure) are the ending events (these terminate the life of an object). The example shows a possible user interaction with the prototyper. In a first step, the user creates a Person object by clicking the "mecrperson" button on the Person tab. The result of this can be seen on the top right, namely that a Person named Felix has been instantiated and is now in the state exists. Next, the user attempts

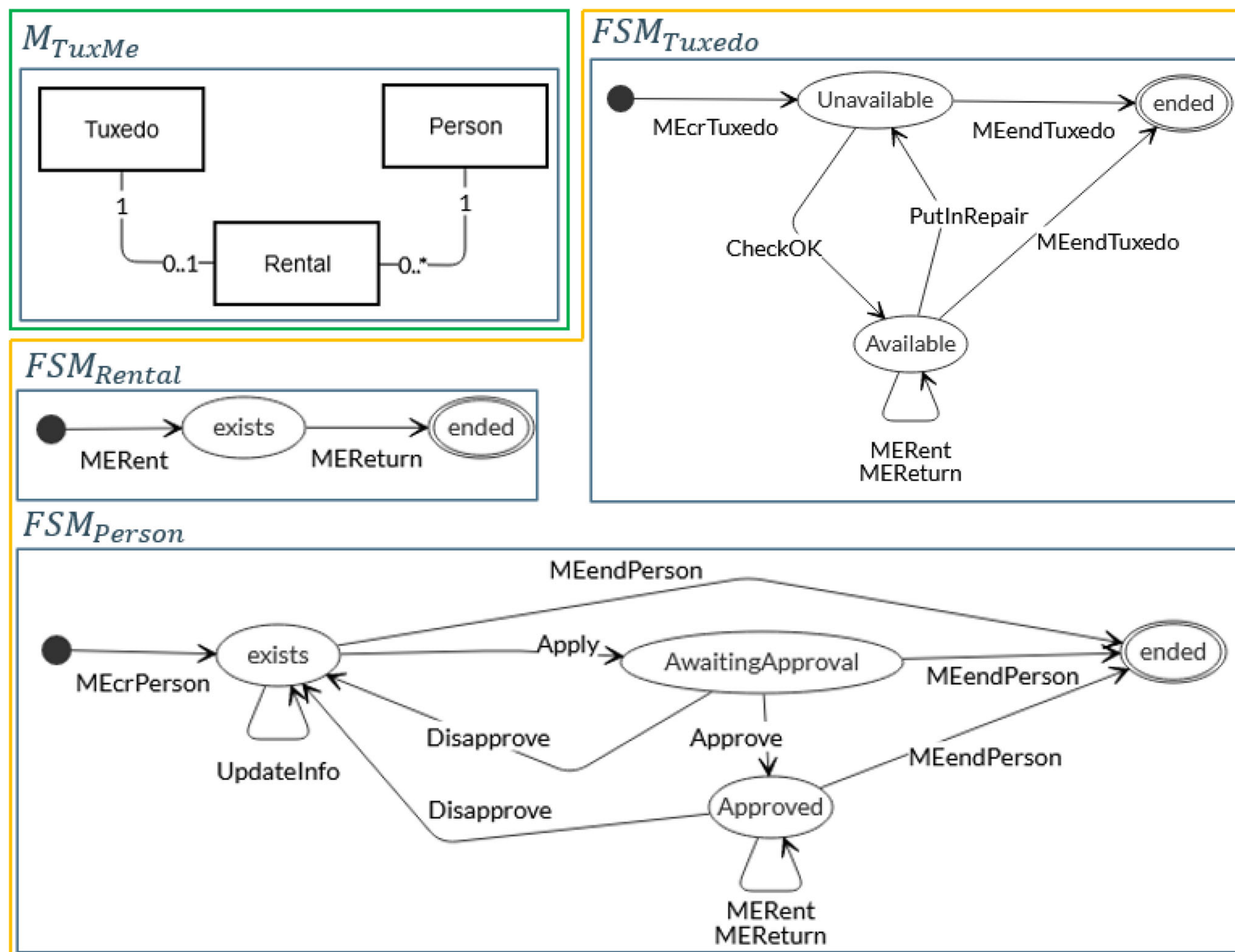


Fig. 2 Example conceptual model

to execute the “approve” event on the Felix object by first selecting the Felix row and then clicking the “approve” button. As soon as the user presses the “approve” button, he is informed that this action is not possible due to a sequence constraint: the Person FSM does not have a transition for the “approve” event in the “exists” state. If the user clicks on “See my FSM”, additional graphical feedback is provided, showing the current state of the Felix object, and a suggested state in which there is a transition for the “approve” event. This type of feedback is negative in nature, as it is only displayed when the user performs an illegal action. This could lead to users being demotivated to test their model thoroughly, as they do not feel a sense of reward after performing correct actions.

2.4 TesCaV

TesCaV is a model-based testing tool that provides users with feedback on their manually executed test cases. This means that, unlike the general MBT process shown in Fig. 1, TesCaV does not concretise the test cases, but after the abstract test cases have been generated (step 2), these generated test cases are used to provide users with feedback on their manual test case execution.

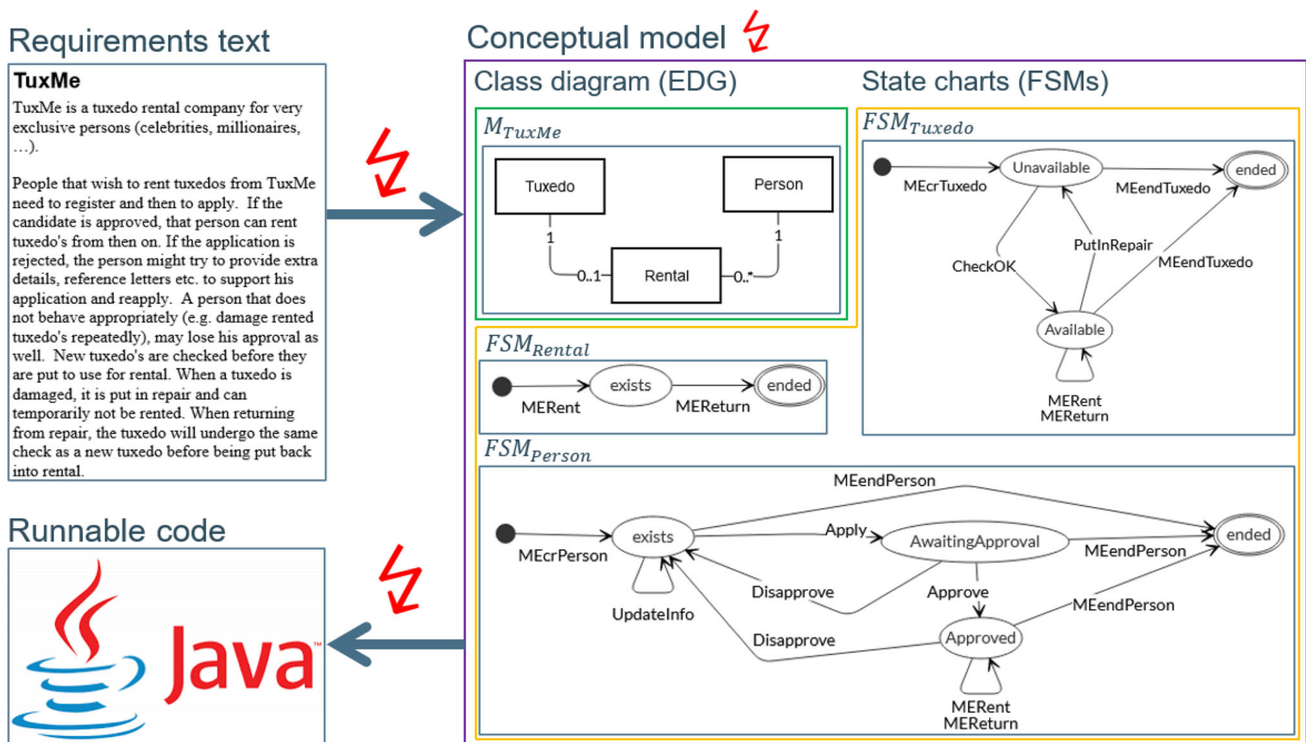


Fig. 3 Overview of the possible discrepancies within the MERODE approach

As input for the generation of abstract test cases, TesCaV uses the conceptual models developed by the modeller. Thus, TesCaV takes as input both the class diagram and the associated FSMs. The test plan used by TesCaV is based on TCGen [17, 20], which is an algorithm for generating test cases based on several criteria, namely the Class Attribute (CA), All-Transitions (AT), All-States (AS), All-Loop-Free-Paths (ALFP), All-One-Loop-Paths (AOLP), Association-End-Multiplicity (AEM), Generalization (GEN), Transition-Pairs (TP), All-Loops (AL) and All-Methods (AM) criteria. These criteria can be divided into four sub-categories: class-based criteria (CA, AEM and GEN), transition-based criteria (AT, ALFP, AOLP, AL), state-based criteria (AS) and method-based criteria (AM). Using both the conceptual model and the TCGen algorithm, TesCaV automatically generates a set of abstract test cases. As all these criteria are based on the structural coverage of software artefacts, TesCaV thus only provides feedback on these structural coverage criteria and does not provide any feedback on integration testing, for example.

To provide feedback on the user's manual test cases, TesCaV uses the event log, which is automatically generated from the user's interactions with the prototyper. These event logs capture the execution of all events in the prototyper and thus represent the full set of manual test cases that the user has attempted to execute with the prototyper. By automatically comparing the generated test cases with the event log, it is possible to check which test cases have been covered by the user and which have not.

Figure 5 shows an overview of the integration between the MERODE approach, the MBT process and TesCaV. As the MERODE approach is an MDE approach, there is already a model, the conceptual model, which can be used to generate test cases in an MBT approach. By automatically comparing the event log generated from user actions and the test cases automatically generated from the conceptual model via the test coverage criteria, a list of covered and uncovered test cases can be generated. This list can be used to give the user feedback on their manual test case execution.

Furthermore, an example of how a user can run TesCaV from within the prototyper is shown in Fig. 6. After clicking on the "Run TesCaV" button, the user is informed about the (in)completeness of his manual test cases. In this case, there are test cases that the user has not yet covered. The user can then ask for more details by selecting an object type on which they would like more feedback. In this case, the user requests feedback on the Tuxedo object type. For each of the different test criteria, the user is then informed of the number of test cases covered and the total number of test cases. For example, the user has covered three of the six test cases for the all-transitions coverage. It is important to note that this feedback is only for the Tuxedo object type and there may be other uncovered test cases for all-transitions coverage for other object types. Finally, the user can request more detailed

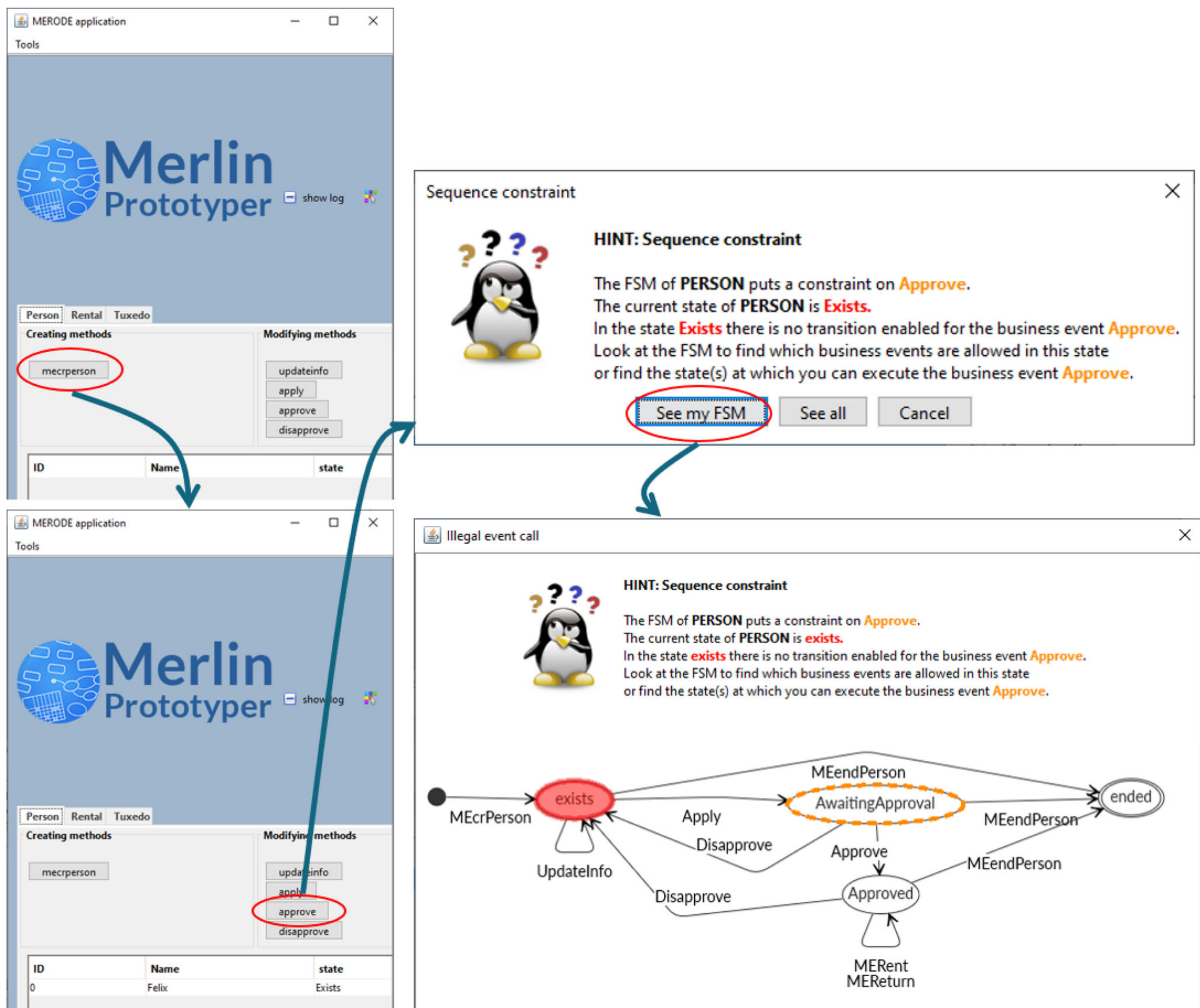


Fig. 4 Overview of the prototyper in the MERODE approach and feedback provided to the user

feedback on any of the criteria by clicking the View button. This provides the user with colour coded visual feedback on the covered and uncovered test cases. The type of feedback provided by TesCaV corresponds to the following feedback types defined in [28]: informative, both positive and negative, and learner driven. As TesCaV also includes positive feedback, namely green coloured components to represent tested parts of the model, users may feel more motivated to test their models thoroughly.

TesCaV can provide feedback on both positive and negative tests. Positive testing checks whether the usage scenarios that should be allowed by the software system are actually allowed. Negative testing checks whether the usage scenarios that should not be allowed by the software system are correctly blocked by the system. Consider the software model shown in Fig. 2. The requirements in Fig. 8e dictate that a person can only rent a tuxedo after applying and being approved. In a positive test, a tester would confirm that an approved person can successfully initiate the rental process by transitioning from the Approved state using the MERent method. Similarly, a positive test would be able to detect a hypothetical missing Approve transition in the AwaitingApproval state. However, relying on positive testing alone would fail to detect a potentially incorrect transition that would allow people to rent a tuxedo before being approved, a scenario that negative testing would detect. Negative testing would also allow a tester to confirm that it is impossible for a person to rent a tuxedo before applying and being approved, meaning that the person would still be in either the Exists or AwaitingApproval state.

TesCaV provides users with feedback on their manual testing efforts for their own software models. This creates

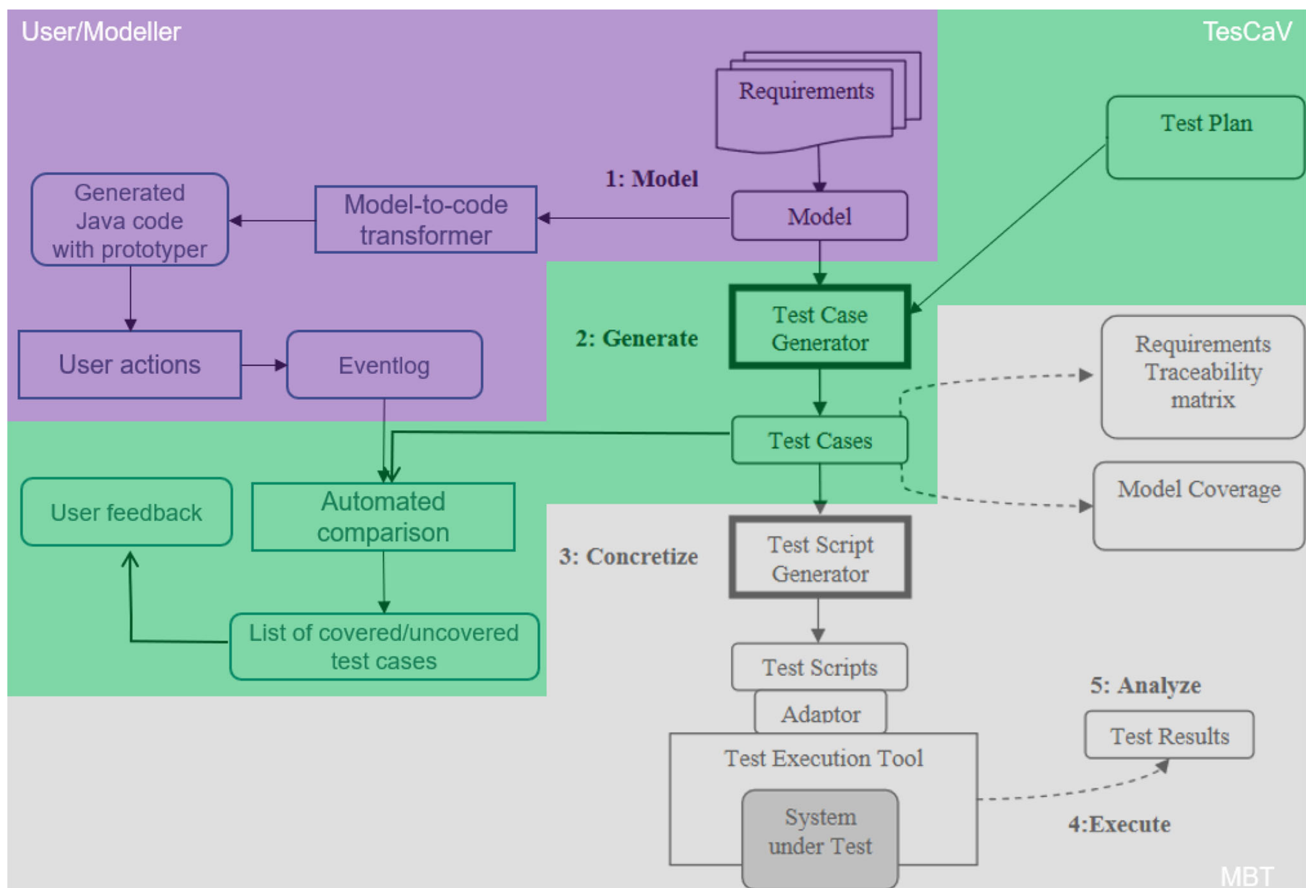


Fig. 5 Integration of the MBT process and TesCaV with the modelling and prototyping process

an incentive to execute more events on the prototype application to get more positive feedback. These more thorough checks allow users to validate that all requirements are indeed correctly implemented in their conceptual models. Together with the feedback provided in the prototyper (Fig. 4), this should allow the discrepancy between the requirements and the developed conceptual model (as shown in Fig. 3) to be reduced.

Although there are many model-based testing tools available, they all generate and execute test cases automatically, making it difficult to teach novice users how to perform adequate testing [15]. To our knowledge, TesCaV is the only model-based testing tool that can help novice users understand how to perform effective testing. This is due to the feedback TesCaV provides through its visualisations, allowing novice users to better understand their manual testing efforts. Previous research has shown that TesCaV is beneficial to the learning process of students in the field of software testing, and has been found to lead to more comprehensive testing of software systems when used by students [3, 16]. However, no evidence has been found to suggest that TesCaV has a lasting positive effect on novice users' ability to achieve high test coverage, even after they have stopped using the tool. This research investigated this lasting positive effect by introducing a new experimental design to further evaluate TesCaV.

3 Research method

3.1 Course context

The experiment was conducted during the 2022–2023 academic year in one of the classes of the Architecture and Modelling of Management Information Systems (AMMIS).[1]

AMMIS is a one-semester course offered at the Master's level with an active learning approach. The aim of the AMMIS course is to teach students to build conceptual models based on given requirements. The teaching approach is based on a blended learning approach consisting of on-campus and online lectures and on-campus computer lab sessions. The exercises provided in the computer lab sessions are designed based on the 4C/ID instructional design

[1] More information about this course can be found at https://onderwijsaanbod.kuleuven.be/syllabi/e/D0I71AE.htm.

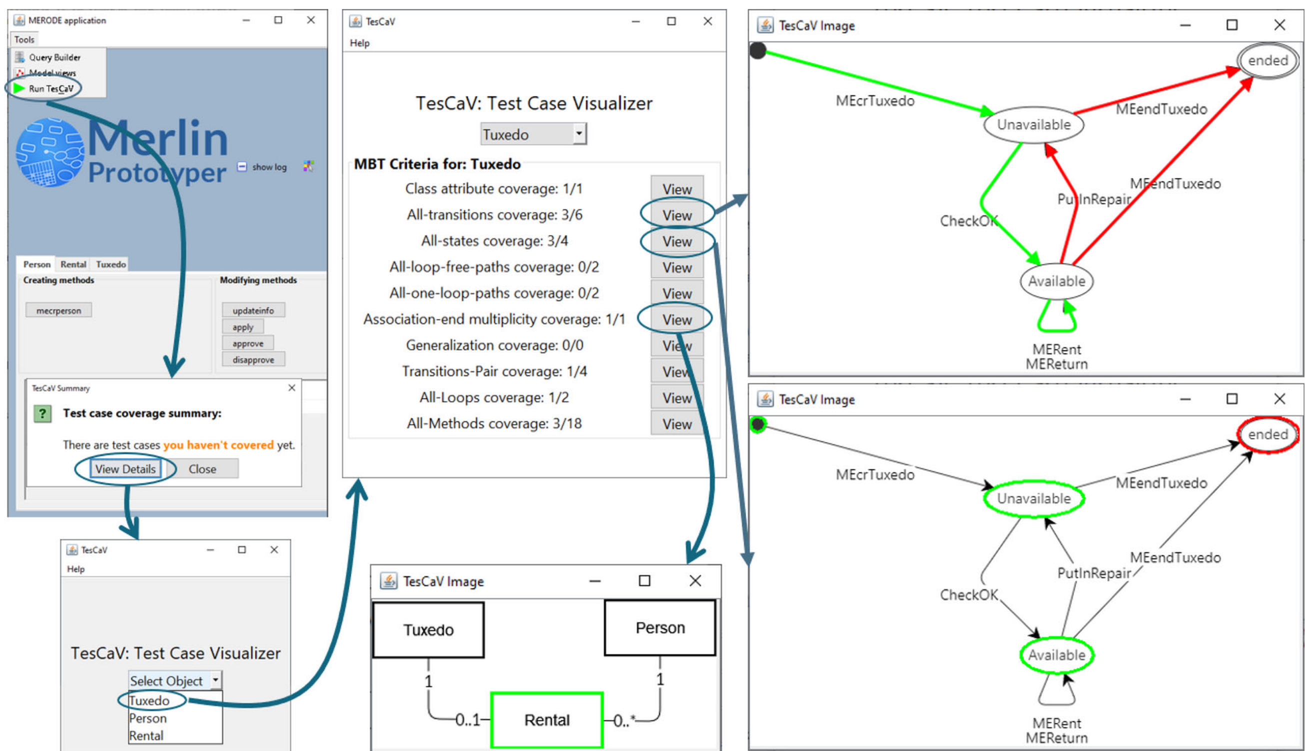


Fig. 6 Overview of the actions needed to perform to get feedback from TesCaV in the prototyper of the MERODE approach

theory, which provides a moderate constructivist approach [32]. This means that the exercises become more complex as the semester progresses and the support provided to students decreases. In practice, at the beginning of the semester, full support consists of giving students pre-made conceptual models for given requirements and asking them to experiment with them to understand the implications of their design choices. Later on, students are given requirements for which they only need to construct the class diagram, and support takes the form of guiding questions and hints until they can construct the class diagram without support. Similarly, to learn state chart modelling, students are given the class diagram alongside the requirements as support. By the end of the semester, students have to develop their own conceptual models from scratch.

3.2 Course integration

As TesCaV is implemented as a module in the MERODE code generator, it can be used in any of the exercises. The variety of exercises leads to a similar variety of what TesCaV's feedback can do for the students. If students are given a complete model that is considered correct with respect to the given requirements, TesCaV's feedback will help students understand the completeness of their manual testing efforts. Further testing will allow students to better understand the design choices of the model they have been given. When students are asked to complete a partial model, TesCaV can provide feedback not only on the completeness of their manual testing efforts, but also on the correctness and completeness of their conceptual model with respect to the requirements. For example, if the requirements state that a person should always be able to rent a tuxedo, the student should manually test that this is indeed possible in every state of the Person object. A student might forget to do this for one of the states, which TesCaV's feedback could inform them about.

3.3 Experimental setup

A well-known theoretical model of learning retention/transfer is Kirkpatrick's which proposes four levels for evaluating training: Level 1 Reaction, Level 2 Learning, Level 3 Behaviour and Level 4 Results [13]. The levels are described in terms of building and evaluating chains of evidence and assessing the extent to which training contributes to outcomes by evaluating whether the results meet expectations.

In the past, we have adapted Kirkpatrick's model to technology-enhanced learning ([22]), where we suggest measuring learning twice: with a test without and with the tool. Suggested measures of the quality of learning are error rates and %completion of the task. This corresponds to the way we approached the design of the experiment, where we will measure model coverage corresponding to %comple-

tion of task. In this experiment, we do not use Kirkpatrick's behavioural level. For this level, the suggestion in [22] is to train as much as possible on real tasks and measure the level of support on real tasks. TesCaV itself works on real-life tasks, but in this experiment, a small case is used to avoid fatigue of the participants. The assessment is therefore a weak assessment of real-life tasks.

Therefore, in order to measure the knowledge transfer effect of test coverage after using TesCaV, a one-group pre-test-post-test design is used. In this experimental design, participants' test coverage is measured before and after using TesCaV, with TesCaV considered as the treatment. Ideally, to avoid some single-group threats to validity, such as instrumentation, history and testing threats, a pre-test-post-test control group design should be used. This is where a second group takes the same pre- and post-test but does not receive the treatment. Unfortunately, in education this leads to the unethical practice of giving students unequal learning opportunities. Although it would be possible to provide the control group with an equivalent alternative treatment, such as an additional lecture on the same software testing topic, it would be difficult to prove that such a treatment is of equal educational value. Furthermore, since participation in the trial cannot be made compulsory, it is difficult to ensure random allocation to experimental groups that receive their treatment at different places and times. Therefore, for ethical and partly for convenience reasons, a pre-test-post-test control group design cannot be used in this case.[2] The limitations of this approach are discussed in more detail in Sect. 5.1.

Before the experiment, the basics of testing are reviewed, such as the V-model, the notion of positive and negative test cases, and the need for a strategy given that exhaustive testing is not possible. The participants are told that their task is to develop test cases for system and acceptance testing: the test cases must assess whether the developed system meets all the requirements. During the pre-experiment questionnaire the personal characteristics of the participants are collected. These questions are given in Table 5 in the appendix. The questions relate to age, gender, confidence in using new computer tools, and previous knowledge of software testing. During the pre-test, participants are asked to use pen and paper to develop a test strategy and an appropriate set of test cases for a given case. The students are given both requirements and models for the case. The treatment is then applied by asking the participants to solve a second case using TesCaV. Again, the students are given the requirements and the model, but this time they are also given the generated prototype application containing the TesCaV module. The students are shown how to get feedback from TesCaV. Finally, during the post-test, students are asked if they would like to make any changes to the test suite they developed for the first case. Students can add new test cases or remove test cases they consider redundant or unnecessary. This is again done on paper, without the use of a generated prototype application.

After each pre-test, treatment and post-test, participants are asked a series of questions to understand their perceived test case coverage. These questions are given in Table 6. An overview of the experimental design is given in Fig. 7.

For Case A (pre-test), the "TuxMe" case was used, while for Case B (treatment) the "PhilHarmonics" case was used. The requirement text for these cases is given in Figs. 8e and 9f, respectively. The software models for these cases are shown in Figs. 8 and 9, respectively. Both cases describe relatively small systems consisting of only a few domain object types. However, Case B is more complex than Case A because it has an additional domain object type and also has more states in the finite state machines associated with its object types. Students are therefore able to experience the feedback provided by TesCaV on a richer model with a more complex structure.

This experimental design allows us to test the following hypotheses. Assuming that after using TesCaV, students will have a better understanding of how to achieve full coverage, students will feel more confident about their performance, but also actually perform better. It is also assumed that the effects of the treatments are the same for all students, regardless of their personal characteristics.

Hypothesis 1 *The usage of TesCaV yields an increase in the perceived test coverage of a software system for novice users in terms of positive and negative test cases.*

Hypothesis 2 *Personal characteristics such as age, gender and previous knowledge on software testing do not affect this increase in perceived test coverage.*

Hypothesis 3 *The usage of TesCaV yields an increase in the actual test coverage of a software system for novice users in terms of positive and negative test cases.*

Hypothesis 4 *Personal characteristics such as age, gender and previous knowledge on software testing do not affect this increase in actual test coverage.*

4 Results

4.1 Demographics

In total, 45 participants completed the questionnaire. Two participants did give informed consent, leaving 43 valid responses. Of the valid responses, 22 were from female participants and 21 were from male participants. The mean age

[2] Ethical approval for this research can be found under number G-2023-6441-R3(MIN).

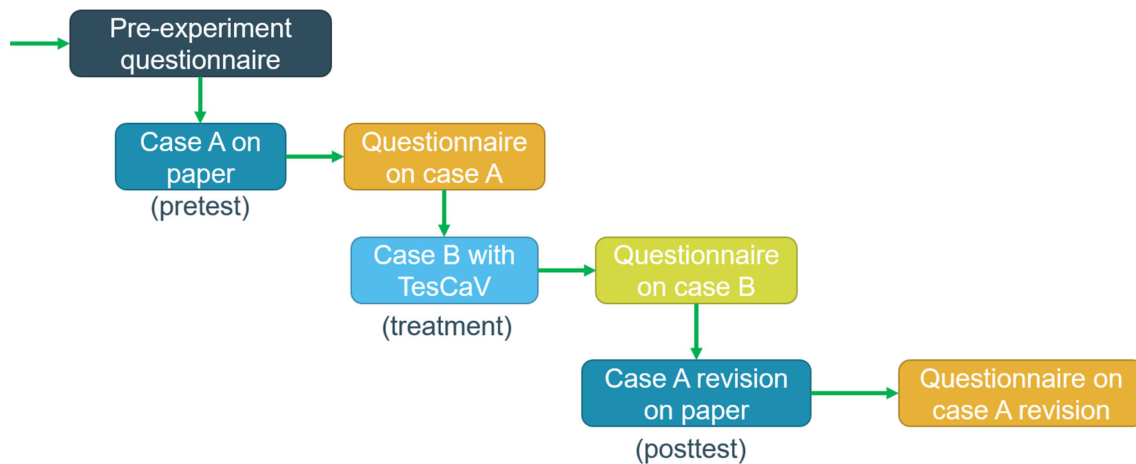


Fig. 7 Experimental setup for the experiment

of the participants was 23.1 years. Unfortunately, not all participants returned their written test cases, so only the results of 37 participants could be analysed for the comparison of Case A before and after treatment.

4.2 Perceived test coverage

This section discusses which results can be used to provide evidence for H1 and H2. Evidence for H1 can be found through an increase in the participants' perceived test case coverage, while evidence for H2 can be found through the lack of correlation between the change in perceived test case coverage and personal characteristics.

Participants completed several Likert scale questions to measure their perception of test coverage, as shown in Table 6. The first two questions measure whether the participants feel that they have identified a sufficient number of test cases for adequate testing, even though they would be able to formulate more test cases. The next two questions measure whether the participants are unable to think of additional test cases, even though they feel that more are needed. These are two different reasons for stopping testing: the first refers to stopping because they feel they have tested enough, the second refers to stopping because they do not know what else to do. Participants answered these questions after both the pre-test and the post-test. The responses were analysed using a coding for the 5-point Likert scale, where "Completely disagree" is 1 and "Completely agree" is 5.

For H1, an overview of participants' perceptions of their test coverage is given in Table 1, which compares students' self-reported scores after the pre- and the post-tests. As the data consist of a measurement taken at two different points in time (after the pre-test and after the post-test), the groups are dependent. A paired t-test would be appropriate. However, as the measurements are on an ordinal scale, i.e. the responses are on a Likert scale, a nonparametric test is required. A Wilcoxon signed-rank test, which is the nonparametric version of the paired t-test, can be used for this comparison. The results of the Wilcoxon signed-rank test are given in Table 1. An additional Wilcoxon signed-rank test was performed on the variables. This additional Wilcoxon signed-rank test works under the alternative hypothesis that case A (pre-test) has a lower mean than case A: revision (post-test). The results are presented in the same table.

For H2, the change (between pre-test and post-test) in the perceived test case coverage can be correlated with the personal characteristics of the participants. This correlation can be calculated by coding the responses to the Likert-scale questions about participants' perceived coverage of test case from 1 ("Completely disagree") to 5 ("nCompletely agree"), and by coding the responses to the confidence and experience questions in the same way. Using these coded scores, Spearman's correlation coefficient can be applied as the data are ordinal (Likert scale). The results are shown in Table 2.

4.3 Actual test coverage

An increase in participants' actual test case coverage by participants provides evidence for H3, while evidence for H4 can be verified by checking that there is no correlation between the change in actual test case coverage and personal characteristics.

For H3, an overview of the students' actual test coverage is given in Table 3, which compares the number of test cases identified by the students during the pre-test and the post-test. Comparing the actual number of test cases identified is equivalent to comparing the test coverage, since to obtain the test coverage the actual number of test cases identified should be divided by the total number of automatically generated test cases based on the model. Since both the pre-test

Table 1 Median of the participants' own **perception** of test coverage for the **pre-test** and the **post-test**

Question	Case A Pre-test (m_1)	Case A revision Post-test (m_2)	Difference	Wilcoxon signed rank test (p value)	
				$H_a : m_1 \neq m_2$	$H_a : m_1 < m_2$
Perceived sufficient positive testing	3	2	− 1	0.59	0.71
Perceived sufficient negative testing	2	2	0	0.23	0.89
Perceived exhaustive positive testing	2	2	0	0.19	0.10
Perceived exhaustive negative testing	2	2	0	0.22	0.11

Table 2 Correlation of personal characteristics with the difference in **perceived** test coverage between the **pre-test** and the **post-test** using the Spearman correlation coefficient (p value)

Variable	Age	Gender	Confidence	Experience
Perceived sufficient positive testing	0.14	0.44	0.74	0.99
Perceived sufficient negative testing	0.46	0.51	0.46	0.80
Perceived exhaustive positive testing	0.59	0.98	0.28	0.07
Perceived exhaustive negative testing	0.45	0.87	0.55	0.19

Table 3 Average number of **actual** test cases identified per participant for the **pre-test** and the **post-test**

Testing type	Case A Pre-test (m_1)	Case A: revision Post-test (m_2)	Difference	Wilcoxon signed rank test (p value)	
				$H_a : m_1 \neq m_2$	$H_a : m_1 < m_2$
Positive test cases	3.75	4.58	0.83	$\mathbf{3.47 \times 10^{-4}}$	$\mathbf{1.73 \times 10^{-4}}$
Negative test cases	3.06	3.88	0.81	$\mathbf{2.08 \times 10^{-4}}$	$\mathbf{1.04 \times 10^{-4}}$
All test cases	6.81	8.46	1.65	$\mathbf{4.51 \times 10^{-5}}$	$\mathbf{2.25 \times 10^{-5}}$

Bold indicates significant p-values

and the post-test used the same model, this would give the same denominator. Similarly to H1, given the ordinal nature of the data, a Wilcoxon signed-rank test was used for this comparison. The results of the Wilcoxon signed-rank test are shown in Table 3. An additional Wilcoxon signed-rank test was performed on the variables. This additional Wilcoxon signed rank test works under the alternative hypothesis that Case A (pre-test) has a lower mean than Case A: revision (post-test). The results are presented in the same table.

For H4, the change (between pre-test and post-test) in the actual coverage of the test cases can be correlated with the personal characteristics of the participants. This correlation can be calculated by encoding the responses to the confidence and experience questions, where "Completely disagree" is equal to 1 and "Completely agree" is equal to 5. Using these coded scores, the Spearman correlation coefficient can be used, as the data are ordinal (count and Likert scale). The results are shown in Table 4.

Table 4 Correlation of personal characteristics with the difference in **actual** test coverage between the **pre-test** and the **post-test** using the Spearman correlation coefficient (p value)

Variable	Age	Gender	Confidence	Experience
Positive test cases	0.29	**0.03**	0.74	0.59
Negative test cases	0.17	0.22	0.26	0.42
All test cases	0.15	**0.04**	0.36	0.73

Bold indicates significant p-values

5 Discussion

For H1, the results in Table 1 show that no significant changes in the responses to the perceived sufficient and exhaustive testing were found for either the positive or negative test cases. This means that no evidence was found for H1.

For hypothesis H2, the calculated Spearman correlation coefficients in Table 2 should be considered. As all the p values obtained are above the 0.05 significance threshold, no correlation was found between any of the personal characteristics and the perceived test coverage variables. This provides evidence for H2.

To test hypothesis H3, Table 3 shows the actual number of test cases identified for the pre- and post-tests. For both positive and negative test cases, and therefore for all test cases, there is an increase in the average. Since all p values of the Wilcoxon signed rank test are significant, it can be said that the means of the pre- and post-tests are statistically different. The additional Wilcoxon signed rank test with the alternative hypothesis that the mean of Case A (pre-test) is smaller than Case A: revision (post-test) also shows that all p values are significant, which means that significantly more

test cases were defined in the post-test than in the pre-test. This provides evidence for Hypothesis H3.

For hypothesis H4, the calculated Spearman correlation coefficients in Table 4 should be considered. Gender is the only variable that correlates with the increase in actual positive tests and all cases. As the other correlation coefficients are all above the 0.05 significance threshold, no other correlations were found to support hypothesis H4 specifically for these personal characteristics.

So even though the participants' perception of the coverage achieved remains the same between the pre-test and the post-test, in reality they have increased their actual test coverage. The fact that this actual increase in test coverage is not reflected in their perception of test coverage could be seen as an indication that their perceptions are now closer to reality. In other words, students are showing less unjustified optimism. This leads us to believe that the use of TesCaV allows participants to understand that testing is a complex task and that they have begun to realise that testing all possible test cases of a software system is unrealistic.

These results also provide some insights for course design. Students who are new to software testing, seem to believe that they are achieving good code coverage on a software system through their manual testing efforts. This is partly because they are not fully aware of how complex the task of fully testing a software system is, and because they are not given feedback on how much their manual testing efforts have contributed to the overall test coverage of a software system. Providing students with a tool that gives them feedback on the coverage achieved by their manual testing efforts can therefore be useful. Such a tool also allows students to be more systematic in their approach, as the results show that the students are able to identify more test cases in the same software system than before the treatment, even when the tool support is no longer provided.

Although TesCaV is now implemented as a module in the MERODE code generator, the general idea of providing visual feedback on a user's manual testing efforts can be applied to any course. Courses using MDE approaches can implement a similar tool, possibly with some slight adaptations according to the modelling artefacts they use. These adaptations would be in the visualisations and in the test case generation algorithms used. Courses teaching code-based software testing can also implement a similar TesCaV tool, where the first step would be to develop a model of the code, which can then be used to perform model-based testing.

5.1 Internal validity

The use of a one-group pre-test-post-test experimental design introduces several threats to validity. To address the instrumentation threat, where participants may encounter different difficulties at pre-test and post-test, leading to changes in scores unrelated to the treatment, the same case was used for both pre-test and post-test. The history threat, where improvements may be due to factors other than the treatment, such as previous software testing courses, was mitigated by assessing personal characteristics, which showed only moderate correlations with scores. In addition, the test threat, also known as priming, was avoided by using the same case for both the pre-test and the post-test.

The use of only one question for each of the characteristics in Table 6 also poses a threat to the internal validity. Only one question per characteristic was used in order to keep the questionnaire as short as possible, given that the duration of the experiment was limited to two hours, and to avoid participant fatigue. The use of Cronbach alpha to measure consistency within these traits is not possible because only one question is asked per trait. Nevertheless, a pre-test was carried out with 5 participants to ensure that all the questions and the whole experimental design were clear.

5.2 External validity

A pilot study was conducted to increase the external validity of the research. The pilot study aimed to obtain qualitative feedback and was subject to time constraints. As a result, only four participants were recruited. This small sample size made statistical analysis of their responses irrelevant. Nevertheless, the qualitative feedback provided proved useful in identifying errors and ambiguities in the experimental design, which led to improvements in the student instructions and questions. In order to strengthen the external validity of this study, it is important to consider the sampling method. Specifically, the experiment recruited students who all took the same course, in the same year and on the same campus. A replication study could strengthen the findings, but would ideally recruit students from different teachers and different campuses, courses and programmes.

Although the relatively tight integration between TesCaV and the MERODE code generator, as shown in Fig. 5, would suggest a low generalisability of the results, we are convinced that the results of this research are still generalisable. Namely, the general idea of providing feedback on manual test execution performed by novice testers can be helpful for all types of testing. Specifically for MBT, other MDE approaches that are not per se based on class diagrams and statecharts may also benefit from using such feedback to users. In these cases, other test case generation algorithms may be used. The way in which the feedback is visualised can also be modified, depending on the exact artefacts that the MDE approach uses. Other research has already done a comparison of different visualisations for the feedback generated by TesCaV [2]. Unfortunately, these results were not used in the experiment as the research was still under review at the time of the experiment. More generally, for code-based testing approaches, it may also be useful to provide visual feedback on the coverage of the abstract test cases as generated by MBT. However, this would require the user to first create a model from the code.

6 Conclusion

The effectiveness of TesCaV, a tool designed to provide feedback on tests performed on a software model, was further evaluated in this research. TesCaV is primarily aimed at helping novice users to perform adequate testing. To measure the transfer of testing skills after using TesCaV, a one-group pre-test-post-test experimental design was used. Each participant was given a pre-test in which they were asked to write down their test cases, after which they were given the treatment of using TesCaV. Finally, participants were given the opportunity to make adjustments to their first case in which they could not use TesCaV. This allowed the increase in test coverage scores due to the introduction of TesCaV to be measured. A stagnation in perceived test case coverage was observed. Together with the fact that the participants believed that they would not be able to identify any more test cases manually, we believe that TesCaV allows users to better understand that testing is a complex task and that complete testing of a software model is intractable. In terms of actual test case coverage, there was an increase in the extent to which participants were able to increase their actual test coverage when they stopped using TesCaV. In addition, an analysis of the personal characteristics of the participants showed some moderate correlations, in particular between gender and the number of positive test cases identified, and consequently also between gender and the number of all test cases identified. No other correlations were found in the transfer of testing skills based on different personal characteristics.

6.1 Further research

Future research could focus on conducting a replication experiment to provide additional evidence on the transfer of testing skills from novice users in identifying test cases when using TesCaV. The present experiment measured the increase in test case coverage immediately after using TesCaV, but a new experiment could assess the transfer of testing skills after a longer period of time. In addition, future researchers could compare the study results of students who actually used TesCaV during the semester and those who did not.

Other research [2] has explored the most effective way for TesCaV to provide feedback to users by using different visualisations for the same underlying feedback. These new visualisations can be integrated into TesCaV and evaluated for their ability to further increase test case coverage for novice users. An A/B test could be conducted where one group is given the current version of TesCaV and another group is given the new visualisations to evaluate their effectiveness.

Acknowledgements This paper is being funded by the ENACTEST Erasmus+ Project Number 101055874.

Appendix

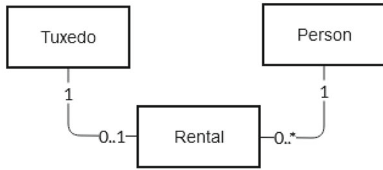
See Tables 5, 6 and Figs. 8, 9.

Table 5 Questions on personal characteristics of the participants given during the pre-experimental survey

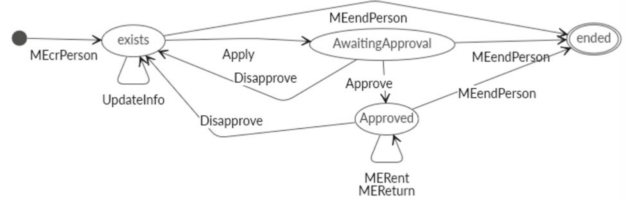
Personal characteristics questions	Possible answers
What is your age?	Number
What is your gender?	M/F/X/Prefer not to say
I have a lot of previous knowledge on behavior modeling/statecharts from a previous degree	no knowledge/experience at all little knowledge (a few hours course) moderate knowledge (intermediate level course) extensive knowledge (advanced course(s))
I have a lot of previous knowledge on programming from a previous degree	no knowledge/experience at all little knowledge (a few hours course) moderate knowledge (intermediate level course) extensive knowledge (advanced course(s))
I have a lot of previous knowledge on testing a software from a previous degree	no knowledge/experience at all little knowledge (a few hours course) moderate knowledge (intermediate level course) extensive knowledge (advanced course(s))
Years of programming experience (if applicable)	Number
I could use a new software application well even if I had never used an application like it before.	Not at all confident: no, probably not, rather not, rather yes, likely yes, totally confident: yes
I could use a new software application well if I had just the built-in-help facility or manual for assistance.	Not at all confident: no, probably not, rather not, rather yes, likely yes, totally confident: yes
I could use a new software application well if I had first seen someone else using it before trying it myself.	Not at all confident: no, probably not, rather not, rather yes, likely yes, totally confident: yes
I could use a new software application well using only the internet for assistance.	Not at all confident: no, probably not, rather not, rather yes, likely yes, totally confident: yes
On average, I use computers (laptops, desktop, tablet) per day	Less than one hour, one to two hours, six to eight hours, eight or more hours

Table 6 Questions given to participants after case A, case B and case A: revision

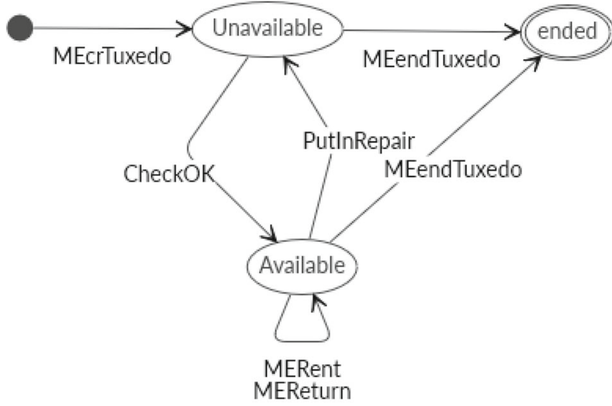
Name	Personal characteristics questions	Possible answers
Perceived sufficient positive testing	I feel like I listed all required test cases to test what should be allowed by the system.	Completely agree
		Agree
		Neutral
		Disagree
		Completely disagree
Perceived sufficient negative testing	I feel like I listed all required test cases to test what should NOT be allowed by the system.	Completely agree
		Agree
		Neutral
		Disagree
		Completely disagree
Perceived exhaustive positive testing	I could not think of any more allowed test cases to test	Completely agree
		Agree
		Neutral
		Disagree
		Completely disagree
Perceived exhaustive negative testing	I could not think of any more disallowed test cases to test	Completely agree
		Agree
		Neutral
		Disagree
		Completely disagree



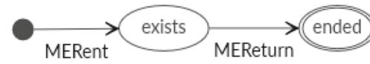
(a) Class diagram for TuxMe.



(b) FSM for Person object type.



(c) FSM for Tuxedo object type.



(d) FSM for Rental object type.

TuxMe is a tuxedo rental company for very exclusive persons (celebrities, millionaires, ...). People that wish to rent tuxedos from TuxMe need to register and then to apply. If the candidate is approved, that person can rent tuxedo's from then on. If the application is rejected, the person cannot rent Tuxedo's. The person might try to provide extra details, reference letters etc. to support his application and reapply. A person that does not behave appropriately (e.g. damage rented tuxedo's repeatedly), may lose his approval as well. New tuxedos are checked before they are put to use for rental. When a tuxedo is damaged, it is put in repair and can temporarily not be rented. When returning from repair, the tuxedo will undergo the same check as a new tuxedo before being put back into rental.

(e) Requirements text for TuxMe case.

Fig. 8 Software model and requirements text for TuxMe case

(a) Class diagram for PhilHarmonics.

(b) FSM for Job object type.

(c) FSM for Application object type.

(d) FSM for Review object type.

(e) FSM for Interview object type.

In the context of recruitment applicants may apply for job vacancies via an Internet online form. Before an applicant can send her application to the respective company, specific information (e.g., name, e-mail address, birthday, residence) must be provided. Once the application has been submitted, the responsible personnel officer in the human resource department is notified. The overall process goal is to decide which applicant shall get the job. Since many applicants may apply for a vacancy, usually, different personnel officers handle the applications.
If an application is ineligible, the applicant is immediately rejected. Otherwise, personnel officers may request internal reviews for each applicant. Depending on the concerned functional divisions, the concrete number of reviews may differ from application to application. Corresponding review forms have to be filled by employees from functional divisions until a certain deadline. Employees may either refuse or accept the requested review. In the former case, they must provide a reason. Otherwise, they make a proposal on how to proceed; i.e., they indicate whether the applicant shall be invited for an interview or be rejected. In the former case an additional appraisal is needed. After the employee has filled the review form, she submits it to the personnel officer. In the meanwhile, additional applications may have arrived; i.e., different reviews may be requested or submitted at different points in time. In this context, the personnel officer may flag already evaluated reviews. The processing of the application proceeds while corresponding reviews are created; e.g., the personnel officer may check the CV and study the cover letter of the application. Based on the incoming reviews he makes his decision on the application or initiates further steps (e.g., interviews or additional reviews). Further, he does not have to wait for the arrival of all reviews; e.g., if a particular employee suggests hiring the applicant.

(f) Requirements text for PhilHarmonics case.

Fig. 9 Software model and requirements text for PhilHarmonics case

References

1. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T.: Reversible Debugging Software. Judge Bus. School, Univ. Cambridge, Tech. Rep 229 (2013)
2. Cammaerts, F., Snoeck, M.: Comparing different visualizations for feedback on test execution in a model-driven engineering environment. In: International Conference on Business Process Modeling, Development and Support, pp. 312–326. Springer, Berlin (2023)
3. Cammaerts, F., Verbruggen, C., Snoeck, M.: Investigating the effectiveness of model-based testing on testing skill acquisition. In: Proceedings of the Practice of Enterprise Modeling: 15th IFIP WG 8.1 Working Conference, PoEM 2022, London, UK, November 23–25, 2022, pp. 3–17. Springer, Berlin (2022)
4. Carver, J.C., Kraft, N.A.: Evaluating the testing ability of senior-level computer science students. In: 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T), pp. 169–178. IEEE (2011)
5. Chan, F.T., Tse, T.H., Tang, W.H., Chen, T.Y.: Software testing education and training in Hong Kong. In: Fifth International Conference on Quality Software (QSIC'05), pp. 313–316. IEEE (2005)
6. Cordova, L., Carver, J., Gershmel, N., Walia, G.: A comparison of inquiry-based conceptual feedback vs. traditional detailed feedback mechanisms in software testing education: an empirical investigation. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, pp. 87–93 (2021)
7. Cowling, T.: Stages in teaching software testing. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 1185–1194. IEEE (2012)
8. Edwards, S.H.: Teaching software testing: automatic grading meets test-first coding. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 318–319 (2003)
9. Fraser, G., Gambi, A., Kreis, M., Rojas, J.M.: Gamifying a software testing course with code defenders. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, pp. 571–577 (2019)
10. Garousi, V., Mathur, A.: Current state of the software testing education in North American academia and some recommendations for the new educators. In: 2010 23rd IEEE Conference on Software Engineering Education and Training, pp. 89–96. IEEE (2010)
11. Garousi, V., Rainer, A., Lauvås Jr, P., Arcuri, A.: Software-testing education: a systematic literature mapping. J. Syst. Softw. **165**, 110570 (2020)
12. Graham, D., Black, R., Van Veenendaal, E.: Foundations of Software Testing ISTQB Certification. Cengage Learning (2021)
13. Kirkpatrick, D.L., Craig, R.L.: Evaluation of Training. Evaluation of Short-Term Training in Rehabilitation, p. 35 (1970)
14. Krasner, H.: The cost of poor software quality in the US: a 2020 report. In: Proceedings of the Consortium Information Software QualityTM (CISQTM)
15. Li, W., Le Gall, F., Spaseski, N.: A survey on model-based testing tools for test case generation. In: Tools and Methods of Program Analysis: 4th International Conference, TMPA 2017, Moscow, Russia, March 3–4, 2017, Revised Selected Papers 4, pp. 77–89. Springer, Berlin (2018)
16. Marín, B., Alarcón, S., Giachetti, G., Snoeck, M.: 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, pp. 302–317. Springer, Berlin (2020)
17. Marín, B., Gallardo, C., Quiroga, D., Giachetti, G., Serral, E.: Testing of model-driven development applications. Softw Qual. J. **25**(2017), 407–435 (2017)
18. Marín, B., Vos, T.E.J.., Paiva, A.C.R., Fasolino, A.R., Snoeck, M.: ENACTEST-European innovation alliance for testing education. In: RCIS Workshops (2022)
19. Martinez, A.: Use of JiTT in a graduate software testing course: an experience report. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, pp. 108–115 (2018)
20. Pérez, C., Marín, B.: Automatic generation of test cases from UML models. CLEI Electron. J. **21**, 1 (2018)
21. Rojas, J.M., Fraser, G..: Code defenders: a mutation testing game. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 162–167. IEEE (2016)
22. Ruiz, J., Snoeck, M.: Adapting Kirkpatrick's evaluation model to technology enhanced learning. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 135–142
23. Strategic Planning: The Economic Impacts of Inadequate Infrastructure for Software Testing, p. 1. National Institute of Standards and Technology (2002)
24. Sarkar, A., Bell, T.: Teaching black-box testing to high school students. In: Proceedings of the 8th Workshop in Primary and Secondary Computing Education, pp. 75–78 (2013)
25. Scatalon, L.P., Carver, J.C., Garcia, R.E., Barbosa, E.F.: Software testing in introductory programming courses: a systematic mapping study. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, pp. 421–427 (2019)
26. Sedrakyan, G., Snoeck, M.: Lightweight semantic prototyper for conceptual modeling. In: Advances in Conceptual Modeling: ER 2014 Workshops, ENMO, MoBiD, MReBA, QMMQ, SeCoGIS, WISM, and ER Demos, Atlanta, GA, USA, October 27–29, 2014. Proceedings 33, pp. 298–302. Springer, Berlin (2014)
27. Sedrakyan, G., Snoeck, M., Poelmans, S.: Assessing the effectiveness of feedback enabled simulation in teaching conceptual modeling. Comput. Educ. **78**(2014), 367–382 (2014)
28. Serral Asensio, E., Ruiz, J., Elen, J., Snoeck, M.: Conceptualizing the domain of automated feedback for learners. In: IberoAmerican Conference on Software Engineering, pp. 223–236. Curran Associates (2019)
29. Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K., Padua-Perez, N.: Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. ACM Sigcse Bull. **38**(3), 13–17 (2006)
30. UML, OMG and I MOF: The unified modeling language UML (2011)
31. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufman, San Francisco (2007)
32. Van Merriënboer, J.J.G., Kirschner, P.A.: Ten Steps to Complex Learning: A Systematic Approach to Four-Component Instructional Design. Routledge, London (2017)
33. Zakaria, Z.: A state of practice on teaching software verification and validation. In: 2009 Annual Conference & Exposition, pp. 14–112 (2009)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Felix Cammaerts is a Ph.D. student at the Research Centre for Information Systems Engineering (LIRIS) at KU Leuven. Actively researching and developing tools to teach students to test and understand their models within Model-Driven Engineering both in terms of validation and verification.

Monique Snoeck holds a Ph.D. in computer science from KU Leuven. She is full professor at the Research Center for Management Informatics (LIRIS), KU Leuven, and visiting professor at the UNamur. She has a strong research track in conceptual modelling, requirements engineering, software architecture, model-driven engineering and business process management. Main guiding research themes are domain modelling, business process modelling, model quality, model-driven engineering, and technology-enhanced learning. Previous research has resulted in the Enterprise Information Systems Engineering approach MERODE, and its companion e-learning and prototyping tool MERLIN and its companion prototyping tool. She is author of 2 books, (co-) author of over 191 peer-reviewed papers. She is associated editor of the BISE journal and (senior) member of the program committee of numerous conferences in the domains of Information Systems such as CAiSE, RCIS, PoEM, and EMMSAD.