

Introducing Test-Driven Modeling to MERODE through a Domain-Specific Testing Language and Automated Model Validation

Dragana Sunaric^[0009–0004–0115–0817] and
Charlotte Verbruggen^[0000–0003–0418–2633] and
Dominik Bork^[0000–0001–8259–2297]

TU Wien, Business Informatics Group, Erzherzog-Johann-Platz 1, 1040 Vienna,
Austria {charlotte.verbruggen,dominik.bork}@tuwien.ac.at

Abstract. Test-Driven Development has proven effective in improving software quality and supporting learning, yet its application to conceptual modeling remains limited. This paper presents an ongoing research project that introduces Test-Driven Modeling into MERODE. It introduces MUnit, a testing framework that includes a DSL for defining model test cases and the required tool support, integrated into the existing modeling tool MERLIN. The tool allows the validation of both structural aspects, represented by Existence Dependency Graphs, and behavioral aspects, represented by Finite State Machines, and provides continuous feedback during model development. This paper presents the design and implementation of the DSL and supporting tooling, followed by an evaluation of MUnit with MERODE instructors. The evaluation focuses on the usability and expressiveness of the developed DSL, and the educational value of using Test-Driven Modeling in modeling education.

Keywords: Test-Driven Development · Test-Driven Modeling · MERODE.

1 Introduction

Model-Driven Engineering (MDE) is a software development paradigm that uses high-level abstract models as primary artifacts during the development lifecycle [7]. By enabling automation through model transformations and code generation, MDE aims to improve productivity and maintainability [19]. Test-Driven Modeling (TDM) introduces concepts from Test-Driven Development (TDD) into MDE [20]. By specifying tests alongside model creation, TDM aims to detect defects early in the modeling and development process, significantly reducing the cost and impact of errors later in the software lifecycle [10].

Testing not only helps evaluate whether software and models are correct but also plays an important role in the learning process. Testing supports iterative thinking, gives students immediate feedback on misconceptions, and helps students learn the semantics of modeling languages more effectively [2, 11]. Allowing students to continuously validate their models encourages them to experiment

and gain a deeper understanding of modeling. This may lead to increased study activity throughout the course, which has been shown to positively impact testing results [15, 16]. Existing research in software engineering education shows that TDD can lead to higher software quality and improve students' productivity, suggesting similar benefits may arise when applying TDD principles to modeling [3]. Despite its potential, TDM is still underexplored, particularly in educational settings. There is a lack of empirical research on the impact of TDM on students' modeling skills.

MERODE is an object-oriented method for enterprise information systems engineering [12]. It is designed to support the development of maintainable and adaptable systems and is currently taught in the courses *Architecture and Modelling of Management Information Systems*, and *Requirements Engineering and Domain Modelling* at KU Leuven. The MERODE toolset includes a code generator that generates a Java application with a user interface based on a given model. This allows users to manually create and interact with model entities through a generated UI. It also provides test coverage feedback, giving users insights into which parts of the model were covered during their manual testing. While the toolset currently supports manual model testing, it lacks functionality to define and continuously execute automated tests on MERODE models as models are created and refined. Prior research has investigated the use of MERODE in educational contexts, focusing on how students interact with executable models and how manual testing, supported by test coverage feedback, can help students perfect their models and improve learning [6, 11, 2]. However, this paper is a first step toward investigating the potential benefits of applying TDM in modeling education. We propose an integration of TDM in the existing modeling approach MERODE, and execute an initial evaluation of its quality and educational value. By introducing automated testing into the modeling process, we will explore whether a fully integrated and automated test-driven approach can improve the modeling workflow for students and provides pedagogical value for model education. The main contribution of this work is MUnit, which consists of a domain-specific language (DSL) for describing test scenarios on MERODE models and asserting expected results, along with the corresponding tool support. The existing MERODE toolset is extended with automated simulation and testing, allowing users to test their models during the modeling process and enabling continuous validation of model behavior. Additionally, we evaluate MUnit's potential to support teaching and learning activities. We aim to answer the following research questions:

- **RQ1:** What are the required components to formalize test cases by means of a DSL for both the Existence-Dependence Graph (EDG) and the Finite State Machine (FSM) aspects of MERODE models?
- **RQ2:** How usable and expressive do MERODE instructors find the developed DSL for defining and executing model test cases?
- **RQ3:** What is the perceived pedagogical value of integrating automated test-driven modeling into MERODE from the perspective of professors and instructors?

2 Background and Related Work

2.1 Software Testing

Software testing plays an important role in ensuring the quality and reliability of software. According to Myers et al. [9], testing begins with the assumption that any program contains errors. This statement generally holds true for almost any software. They define testing as the process of executing a program to find errors. Myers et al. [9] further argue that the quality of a test case is determined by its ability to reveal previously unknown defects. This perspective shifts the goal of the testing process from evaluating correctness to actively seeking faults in the software under test. Martin [5] emphasizes that testing also contributes to the long-term quality of software, since well-designed automated tests improve the maintainability and reusability of production code by providing a safety net during refactoring or incremental development.

The main goal of TDD is to produce clean code that works [1, 5]. In TDD, developers write automated unit tests before writing the corresponding code. This test-first principle ensures that every piece of software starts with an executable specification of its expected behavior. Freeman and Pryce [4] identify several benefits of TDD. They argue that the approach provides continuous feedback on both the quality of the software that is written and the underlying design of the code. Writing tests before writing the functionality forces developers to clarify the acceptance criteria for the next increment of work before implementation begins. This practice also supports the creation of loosely coupled components that can be tested independently and later composed at higher levels. Writing test cases during software development also produces executable descriptions of the system’s intended behavior. Running tests provides immediate feedback by detecting errors while the implementation context is still fresh in the developer’s mind, and it signals when developers have finished implementing a feature.

xUnit testing frameworks provide a standardized architecture for developing and executing automated tests for different domains [8]. Numerous frameworks in different domains follow the xUnit architecture, including JUnit for Java and Pytest for Python.

2.2 Test-Driven Modeling (TDM)

TDM is the adaptation of TDD to conceptual modeling. Similar to TDD in software development, TDM requires test-first thinking and supports incremental refinement, using predefined test cases for continuous validation [8]. However, instead of executable source code, TDM applies these ideas at the model level. Tests are defined before or during model construction [20]. They define both expected and forbidden behavior of the tested model, often in the form of execution traces combined with assertions. This exactly mirrors the role of tests in TDD, where tests are defined not only to verify software but also to specify the intended model behavior [20, 22]. While closely aligned with TDD, TDM has

received comparatively little attention in the literature, and only a small number of tools, conceptual proposals, and empirical studies have been published.

The existing literature identifies several benefits of TDM. It was shown that introducing tests as first-class artifacts in the modeling process supports earlier validation of models [20, 23] as well as safer model evolution [20–22]. TDM improves understanding of model semantics [21–23] and communication with and among stakeholders [21]. Test cases can serve as a shared artifact that is often easier to understand than formal models for domain experts who are not modeling specialists. Some evidence suggests that stakeholders prefer discussing expected behavior using test cases rather than diagrams. This was shown to lead to clearer requirements and fewer misunderstandings between domain experts and modelers [21].

Research on Test-Driven Modeling remains relatively sparse, with only a small number of approaches and tools proposed to date. Existing work evaluates test-driven principles for modeling from different perspectives, including tool support for behavioral modeling [20], using test cases as first-class artifacts for validating and maintaining models [21, 22], and xUnit-style testing of executable schemas [17, 18]. Most of these approaches are implemented as research prototypes and focus on very specific modeling paradigms, lacking a broadly applicable and fully integrated modeling environment. Tort and Olivé propose one of the earliest systematic approaches to testing conceptual schemas using an xUnit-style software testing framework [17, 18]. They introduce the Conceptual Schema Testing Language (CSTL), a domain-specific language for writing automated tests for executable models defined in UML and OCL [18]. CSTL is a textual and procedural language that follows an xUnit style. Test cases written in CSTL are automated, re-executable, and result in one of three specific outcomes (*Pass*, *Fail*, or *Error*). CSTL is a testing language that operates directly on the semantics of models. Test cases are formulated by constructing information base states, executing domain events, and asserting expected outcomes. Currently, CSTL is still a research prototype. CSTL needs executable UML/OCL schemas and an information processor, implemented on top of the USE tool, to run. The work currently does not provide large-scale empirical evaluations, and there is currently no evaluation of educational usages for this type of model testing.

2.3 The MERODE Approach

MERODE is an object-oriented method for Enterprise Information Systems Engineering developed at KU Leuven [12]. It supports the design of enterprise information systems and includes mechanisms for checking formal correctness and consistency of a model. The method is used in both research and education.

MERODE models contain three artifacts that describe the structure and behavior of the application domain [12]. The Existence-Dependency Graph (EDG) models the structural relationships between business object types. The Object-Event Table (OET) defines event ownership and how events affect different object types. Finite State Machines (FSMs) specify the lifecycle of each business object type, including the states and the events that trigger transitions between

them. The combined use of EDGs, FSMs, and OETs enables rigorous verification and consistency checks for models and ultimately enables the automated transformation of models into executable system prototypes.

The MERODE methodology is supported by a range of tools¹. This paper extends MERLIN [13], the web-based modeling tool used to design MERODE domain models. It supports the construction of EDGs, OETs, and FSMs within an integrated interface and provides automated checks for structural and behavioral consistency. The MERODE ecosystem provides strong support for manual validation of domain models. However, the toolset currently offers no support for automated testing. In particular, MERODE lacks tools and support to execute test cases automatically and continuously throughout the modeling workflow.

3 MUnit: Requirements and Conceptual Testing Framework

The previous sections highlighted two key research gaps that motivate this work. First, MERODE currently lacks support for automated testing within the modeling workflow. Second, TDM remains underexplored compared to the extensive research available on TDD. To address these gaps, we propose MUnit, a TDM framework specifically tailored to MERODE. The main requirement for MUnit is to enable executable model tests by providing a dedicated Domain-Specific Language (DSL) for specifying test cases and ensuring its integration into the existing MERODE tooling environment. Furthermore, MUnit is designed for educational use, with a focus on usability, understandability, and seamless integration into existing MERODE modeling workflows for students.

The remainder of this section derives the requirements for MUnit's design and implementation, and explains the conceptual foundations of MUnit. We define what aspects of a MERODE model can be tested, how tests are executed, and how integration within the existing MERODE tools is ensured.

3.1 Testing Targets for MERODE

MUnit must allow testing both the static definition of model elements and the dynamic behavior during execution. In this section, we define the structural and behavioral testing targets that cover all aspects of a MERODE model. Structural tests ensure that the structural components, such as object types, attributes, methods, dependencies, and lifecycle models, are present and correctly specified in the model. We identify four structural test categories, with illustrating examples for the model in Figure 1a. The model represented in this figure is adapted from a modeling exercise for MERODE². This exercise was selected because it covers all aspects of structural and behavioral modeling in MERODE.

¹ see: <https://merode.econ.kuleuven.be/>

² The original exercise is called "Hospital HouseMD" and can be found here: <https://merode.econ.kuleuven.be/pages/cases.html>

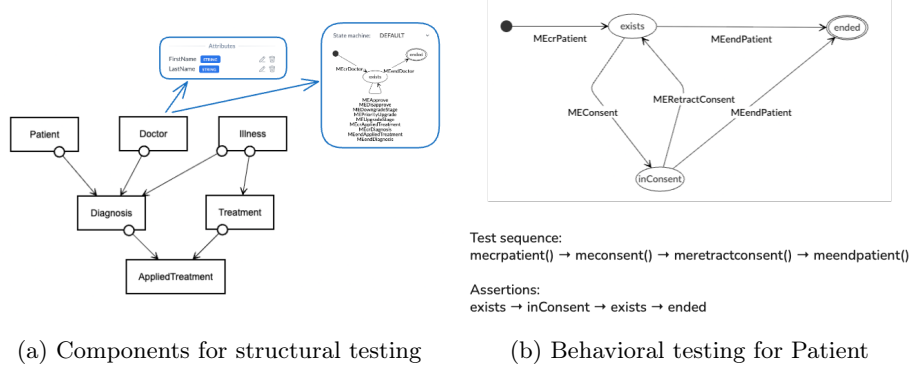


Fig. 1: Example of a MERODE model for testing

- **Structural Element Existence:** The core elements required by the domain model are existent and well-defined. For example, the object type *Doctor* exists in the model and contains the attributes *FirstName* and *LastName*.
- **Structural Object Composition:** An object contains the expected attributes, the lifecycle methods, and the corresponding domain events are available. For example, the Doctor object *doc1* has attributes *FirstName* with value *John* and *LastName* with value *Doe*.
- **Structural FSM Existence and Composition:** The object types' FSM exists and is well-defined. For example, an FSM is defined for the object type *Doctor*; The FSM for *Doctor* contains the states *exists* and *ended*.
- **Existence Dependency:** Verify the correct definition of the expected existence dependency relationships in the model. For example, the existence-dependency relationship between *Doctor* and *Diagnosis* is defined.

Behavioral testing focuses on validating the dynamic aspects of MERODE models, i.e., how objects evolve over time and how events trigger state transitions. Figure 1b illustrates an example object's lifecycle as an FSM and shows a corresponding sequence of events executed by a test case to validate state transitions within that FSM. We identify three behavioral test categories:

- **Object Lifecycle Validation:** Events trigger the intended transitions between states. For example, after triggering *meconsent()* for a patient *p1* in the state *exists*, the patient *p1* is now in the state *inConsent*.
- **Event Sequence Validation:** Correct execution of sequences of events results in valid intermediate states of an object during execution. For example, when executing the test sequence *mecpatient()* → *meconsent()* → *meretractconsent()* → *meendpatient()*, a patient object should be created that visits the states *exists*, *inConsent*, *exists*, and *ended*, in that order.
- **Multiple Object Interaction Validation:** Objects participate correctly in shared events, and their interactions lead to the expected outcomes for all related objects. For example, when a new diagnosis *d1* is created for the patient *p1*, the doctor *doc1* and the illness *i1*, each of these objects (*p1*, *doc1*, and *i1*) remain in their current state, and *d1* is in the state *exists*.

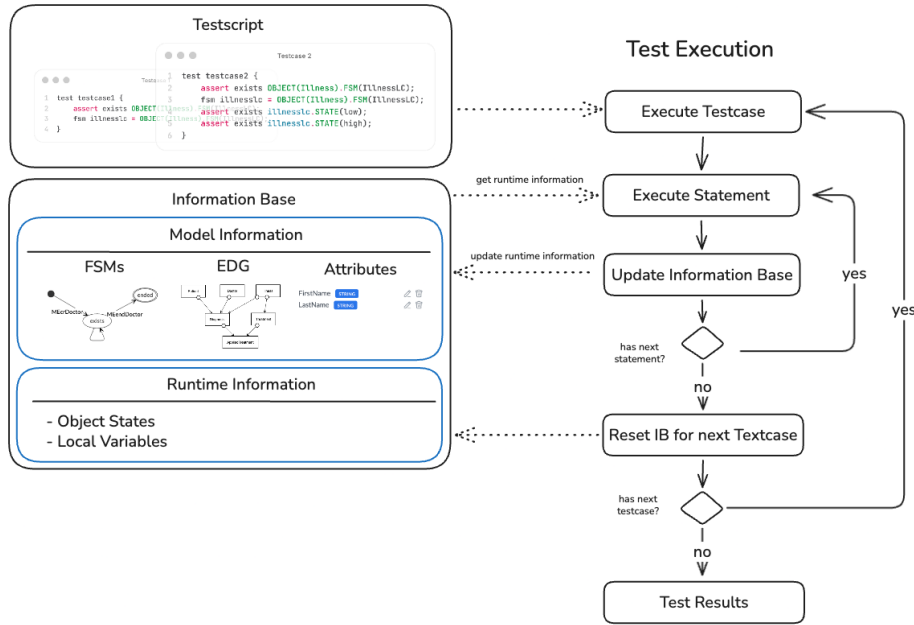


Fig. 2: Conceptual overview of the Information Base, showing its internal structure and its interaction with test case execution and assertion evaluation.

3.2 Information Base

In order to realize the execution and evaluation of test cases for MERODE models in a TDM setting, we use the concept of an *Information Base*, inspired by the Conceptual Schema Testing Language (CSTL) [18]. The Information Base is the abstraction component for maintaining all information and data relevant during test execution. It is a unified repository that contains both the unchanged static structure of the MERODE model and the dynamic state of objects and variables during test execution. Figure 2 shows the structure of the information base and its role during test case execution.

The Information Base provides the execution context for test evaluation, enabling assertions to be evaluated against the current model state while hiding the details of the execution process. Having a single, persistent information base ensures that test cases can be executed in a controlled environment and that assertions can be consistently evaluated against the current state of the model. To ensure independent and reproducible execution, the runtime information in the Information Base is reset before each test case is executed. However, while the runtime state is cleared between independent test cases, the tested MERODE model itself remains unchanged throughout the entire execution. The conceptual structure of the model, including its EDGs, FSMs, and OET definitions, therefore, remains stable across all evaluated test cases.

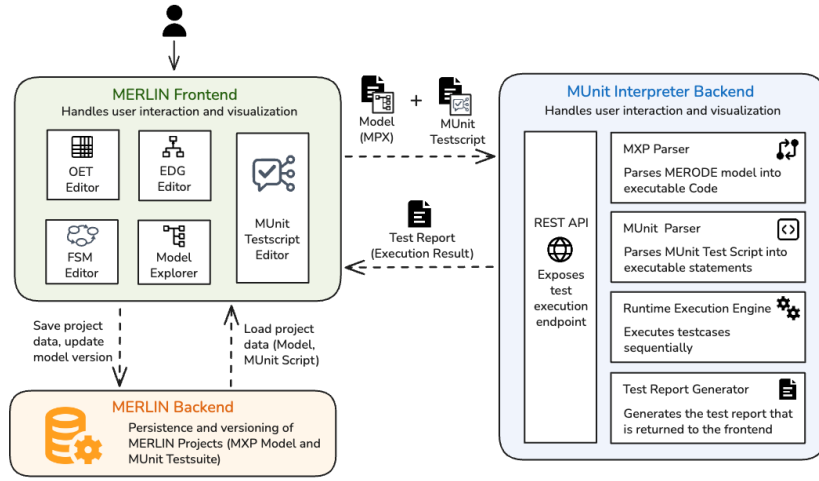


Fig. 3: High-level system architecture diagram

4 Design and Implementation of MUnit

The existing MERLIN Frontend was extended to include an editor for writing test scripts. The MERLIN Backend now also persists the created test script alongside the model. The actual execution of the test suite is delegated to the MUnit Interpreter Backend. The overall architecture and the interactions between the three main components are shown in Figure 3.

4.1 MUnit DSL Design

The MUnit DSL was designed as a textual, external, domain-specific language to specify automated tests for MERODE models. Its syntax follows the style of modern unit-testing frameworks and provides a familiar syntax for students, similar to other object-oriented programming languages.

An MUnit test suite consists of one or more `test` blocks, each representing an independent test case. Test cases are executed sequentially (top to bottom) by the interpreter, and each test is evaluated in isolation from the others. Within a test case, users can declare variables, create object instances, and invoke model events on these objects. Furthermore, users can assign values to local variables and verify expected behavior using assertions.

The concrete syntax of MUnit is formally specified using an ANTLR (Another Tool for Language Recognition) grammar³, which is used to generate the lexer and parser for the MUnit Interpreter. Before the tests can be executed, the parser validates the test script provided as input and transforms it into an Abstract Syntax Tree (AST). The AST is a simplified internal representation of the

³ <https://www.antlr.org/>

test suite independent of the ANTLR parser. The Runtime Execution Engine only uses the generated AST during test execution. Details about the Design and Implementation, including the complete ANTLR grammar, are provided in the online appendix⁴ [14].

Concrete Syntax. Rather than introducing a completely new syntax, MUnit adopts common programming concepts such as variables, method invocations, and assertions while extending them with MERODE-specific constructs for interacting with conceptual models. An MUnit test suite consists of one or more `test` blocks, each representing an independent test case. Test cases are executed sequentially by the interpreter, while each individual test is evaluated in isolation. Test cases can declare variables, create MERODE object instances, invoke model events, assign values to local variables, and verify expected behavior using assertions. This structure closely resembles established unit-testing frameworks, allowing model tests to be written concisely and intuitively.

The language supports two complementary test categories. Behavioral tests validate the dynamic behavior of a MERODE model by creating object instances, executing events, and verifying state transitions or attribute values. Structural tests, on the other hand, verify static properties of the model itself, such as the existence of object types, methods, finite state machines, states, or attributes. These two categories enable automatic validation of both the structural correctness and the behavioral semantics of a MERODE model. Listing 1.1 illustrates a behavioral test case and a structural test case. More examples can be found in the online appendix⁴ [14].

Listing 1.1: Example MUnit behavioral and structural test cases

```

1
2 test behavioral_testIfPatientExistsAfterCreation {
3     Patient p = mecpatient();
4     assert p.getState().getName() != "inConsent";
5 }
6
7 test structural_testExistenceOfCoreObjects {
8     assert exists OBJECT(Patient);
9     assert exists DEPENDENCY(Person, Patient);
10 }
```

The concrete syntax of MUnit is formally specified using a context-free grammar implemented in ANTLR. ANTLR was selected because it provides a concise way to define the language syntax while automatically generating both a lexer and a parser for Java, the implementation language of the MUnit Interpreter.

The grammar defines the syntactic structure of all valid MUnit test suites, including test case declarations, statements, assertions, variable declarations, and expressions. The complete grammar is provided in the Online Appendix⁴ [14].

⁴ <https://github.com/dsunaric/POEM-MUnit-Online-Appendix>

During test execution, the generated parser validates the input script against this grammar before constructing an Abstract Syntax Tree (AST). Syntax errors are therefore detected before any test execution begins, preventing malformed test specifications from entering the execution pipeline.

Figure 4 illustrates the role of the grammar within the interpreter. A textual MUnit test suite is first tokenized by the ANTLR lexer and subsequently parsed according to the grammar. The resulting Abstract Syntax Tree serves as the input for the Runtime Execution Engine.

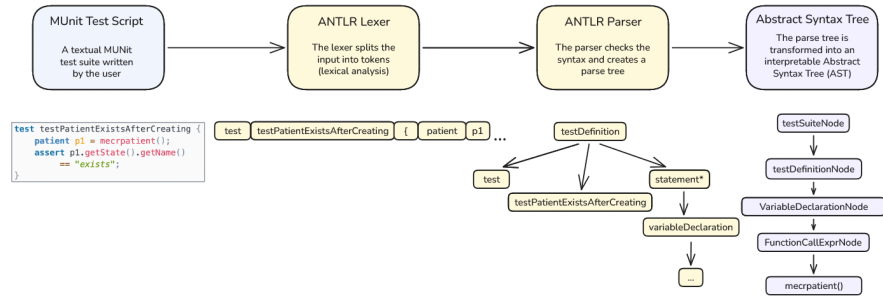


Fig. 4: Transformation of an MUnit test suite into an AST using ANTLR.

Abstract Syntax Model. After successful parsing, the textual representation of an MUnit test suite is transformed into an Abstract Syntax Tree (AST). Whereas the parse tree generated by ANTLR closely reflects the concrete grammar, the AST provides a simplified representation that captures only the semantic information required during execution. Figure 5 presents the class hierarchy of the implemented AST. The model is organized around a small number of abstract interfaces that represent the different categories of language constructs. Concrete node classes specialize these interfaces and represent the individual elements of the MUnit language.

The root of every parsed test specification is represented by the **TestSuiteNode**, which contains the complete test suite. A test suite consists of one or more **TestDefinitionNode** instances, each representing an individual **test** block from the source file. Every test definition contains an ordered sequence of statements that are executed sequentially by the Runtime Execution Engine. Statements are represented through the **StatementNode** interface, which is specialized by the three executable statement types supported by the language. **VariableDeclarationNode** represents variable declarations, **ExpressionStatementNode** models standalone expressions such as method invocations, and **AssertNode** represents assertions that are evaluated during test execution. Expressions are represented independently through the **ExpressionNode** hierarchy. This hierarchy contains the language constructs required to access runtime values, including identifiers, literal values, member access expressions, and function calls. Boolean

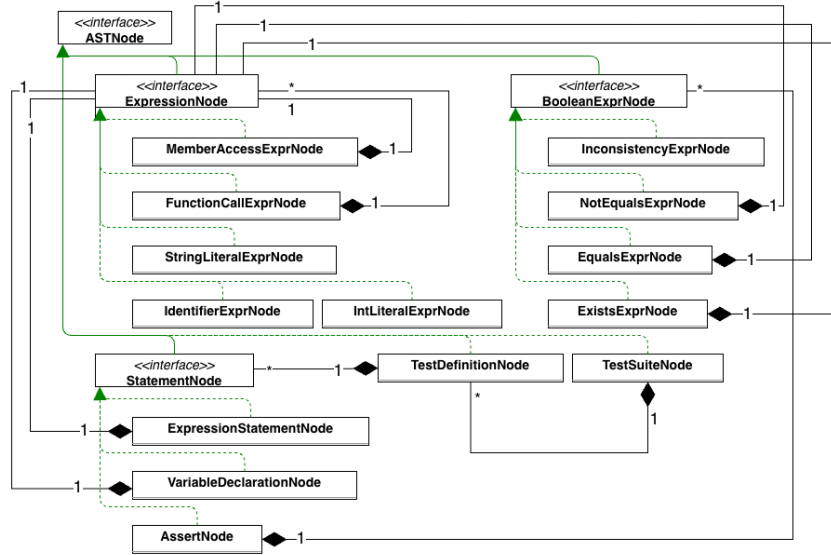


Fig. 5: Class hierarchy of the AST used by the MUnit Interpreter.

conditions used by assertions are represented by the dedicated `BooleanExprNode` hierarchy. The implementation distinguishes among comparison expressions (e.g., equality and inequality checks), structural assertions using the `exists` operator, and consistency checks that validate properties of the MERODE model.

The AST is constructed using the visitor pattern. During parsing, the ANTLR-generated parse tree is traversed by a dedicated `ASTBuilder`, which creates the corresponding AST node for each grammar production. This transformation decouples the execution engine from ANTLR-specific parser classes, resulting in a cleaner architecture and allowing future modifications to the grammar without affecting the runtime implementation.

Answer to RQ1 We identified seven testing targets for MERODE, and developed MUnit, a framework that consists of a DSL for MERODE test cases and an extension of the MERLIN environment to support the creation and automated execution of these tests. The MUnit DSL enables writing test cases for each identified target.

5 Evaluation

To evaluate MUnit, a qualitative user study was conducted with five MERODE experts who actively teach it at three European universities. All participants had multiple years of experience with MERODE and with teaching MERODE, while their experience with automated testing varied. The following sections provide an overview of the study design and results.

5.1 Study Design

The evaluation study consists of interactive sessions with five MERODE educators. Each session started with a brief introduction to MUnit, after which participants completed four guided exercises covering structural testing, behavioral testing, model debugging, and test debugging. The order of the exercises ensured that features of the MUnit language are progressively introduced. The aim was to evaluate MUnit’s usability and the DSL’s expressiveness in realistic modeling scenarios. Rather than asking participants to write test suites from scratch, the exercises consist of writing a test case similar to a given example or fixing one with seeded errors. The aim was to allow participants to focus on understanding the language and the feedback provided rather than on learning the DSL’s syntax during the session.

After the practical exercises, all participants completed a questionnaire providing feedback on the MUnit’s usability and the DSL’s expressiveness. Since all participants have experience in teaching MERODE, they were also asked to assess the educational value of MUnit and its suitability for integration into MERODE courses. During the evaluation sessions, all participants were encouraged to think aloud, while observational notes were recorded to complement the questionnaire results. The questions in the questionnaire were categorized into: Usability, Expressiveness, Educational Value, and Course Integration. The questionnaire statements were answered using a five-point Likert scale ranging from *Strongly agree (1)* to *Strongly disagree (5)*. In addition, the participants were also given the opportunity to provide qualitative feedback through several open-ended questions. A detailed description of the exercises as well as the full questionnaire is included in the online appendix⁴ [14].

5.2 Results

This section presents the findings of the evaluation. The results combine quantitative questionnaire responses with qualitative feedback to assess MUnit’s usability, expressiveness, and educational potential.

Table 1 summarizes the mean ratings for the four questionnaire categories. The responses to the individual questions can be found in the online appendix⁴ [14]. Overall, MUnit was evaluated positively by all participants across all categories. Participants considered the MUnit language easy to learn and sufficiently expressive for specifying test cases for MERODE models. In particular, they agreed that the DSL allows testing of EDGs and FSMs, and is straightforward to learn after only a short introduction.

The educational value of MUnit received the strongest agreement. Participants confirmed that automated testing could improve students’ understanding of MERODE, support earlier detection of modeling errors, and contribute to higher model quality. Likewise, they considered MUnit well-suited for integration into existing MERODE courses, particularly for practical exercises and assignment evaluation.

Table 1: Mean questionnaire ratings by evaluation category.

Evaluation Category	Mean
Usability	2.00
Expressiveness	1.65
Educational Value	1.45
Course Integration	1.60

Table 2: Overview of the main themes identified during the qualitative analysis.

Theme	Summary
Perceived Benefits	Immediate feedback, rapid model validation, reusable test scripts, and testing directly on MERODE models without code generation.
Language and Usability	Suggestions to simplify the DSL through clearer terminology, more intuitive syntax, and reduced dependence on Java-specific concepts.
Editor and Diagnostics	Requests for auto-completion, syntax validation, improved error messages, and richer execution feedback.
Additional Features	Support for multiple test scripts, undo functionality, and extensions to existing assertions.

The only aspect that received comparatively lower ratings was the understandability of the generated error messages. Although participants generally considered the testing workflow intuitive, they indicated that error messages could be more descriptive and are especially hard to understand for students with little knowledge in either MERODE or programming. This is also supported by the qualitative findings.

The qualitative feedback complemented the questionnaire results by giving additional insights into participants’ experiences while using the tool. Comments collected during the think-aloud sessions and the open-ended questionnaire responses were grouped into common themes. An overview of the identified themes is given in Table 2.

Participants consistently identified the immediate execution of tests within MERLIN while creating and refining the model as the greatest strength of MUnit. The ability to seamlessly validate models without generating executable code was perceived as a considerable improvement over the existing workflow and was considered highly valuable for supporting iterative model development. Participants also repeatedly mentioned the educational potential of immediate feedback, noting that students could experiment with their models through rapid trial and error while developing a better understanding of MERODE concepts.

Most suggestions for improving MUnit focused on the editing experience rather than on concepts themselves. Support for auto-completion, real-time syntax validation, and more descriptive, simpler-to-understand error messages were the most frequently mentioned areas for improvement. Participants also proposed several refinements to the DSL, including clearer terminology and a simpler syntax for non-technical users. Overall, the qualitative feedback indicates that MUnit was well-received and identifies several opportunities to further improve the usability of the integrated development environment.

Answer to RQ2 Overall, MUnit scored well on usability and expressiveness. Participants appreciated the ability to validate models directly in the editor. The main opportunity to improve usability is to make the feedback messages more descriptive.

Answer to RQ3 Participants strongly valued MUnit’s pedagogical value, agreeing that it enhances MERODE understanding, enables earlier error detection, and improves model quality. They find that MUnit is well-suited for integration into MERODE courses.

6 Discussion

Until now, validation of MERODE models has been performed manually after model development, making repeated validation during development cumbersome. MUnit’s design and integration within the MERLIN modeling environment support an iterative modeling workflow in which tests can be executed continuously while the model evolves. This enables immediate feedback and allows modelers to iteratively refine the conceptual model and its expected behavior, supporting early detection of modeling errors. Such a workflow closely aligns with the principles of TDD and TDM, where validation and refinement occur continuously throughout the development process.

From an educational perspective, embedding testing directly into MERLIN also offers several advantages. Students receive immediate, continuous feedback on the correctness of their models. Furthermore, MUnit is expected to encourage experimentation by enabling students to quickly run tests after modifying specific parts of the model. Because students remain in a single familiar environment, the cognitive overhead of switching between multiple tools is expected to be lower than in the previous workflow of generating a prototype and manually testing it. For students, this would lower the barrier for testing their models and support a more interactive and iterative learning experience.

Overall, the proposed DSL and its integration within MERLIN were well received by the teaching staff during the evaluation. The evaluation suggests that MUnit’s educational potential is promising. All participants identified the immediate feedback provided within MERLIN as one of its greatest strengths. The resulting short feedback cycles are expected to help students develop a better understanding of modeling concepts and the behavior of MERODE models. The participants’ assessment of MUnit’s potential is also consistent with previous work on Test-Driven Learning and other research on the educational benefits of integrating testing into the development process (see Section 2). Based on these findings, we propose that automated testing should become a standard part of MERODE modeling rather than an activity performed only after model completion.

Limitations. Firstly, the evaluation was conducted with only five participants, limiting the general applicability of the findings. The focus was to select participants who were highly experienced in using and teaching MERODE and well-suited to assess the MUnits’ usability and educational relevance. The evaluation focused exclusively on MERODE course instructors, reflecting their experience with how students will perceive MUnit rather than evaluating it with students themselves or measuring learning outcomes. Secondly, MUnit applies only to MERODE models within the MERLIN environment, and further research is needed to investigate how Test-Driven Modeling can be extended to other conceptual modeling methods and tools.

Future Work. The main direction for future work is to evaluate MUnit in a classroom setting with students, focusing on how it affects students’ results and how they perceive its benefits. The evaluation also identified potential technical improvements. It was mentioned that the usability could be further improved by adding auto-completion. Students also need more support in finding code errors, with error highlighting in the editor, and in improving wording and addressing common error messages. Furthermore, this evaluation focused only on experienced MERODE instructors; investigating its impact on students’ perceptions of MUnit and their learning outcomes would provide further insight into the MUnit’s educational value. Finally, although this work targets MERODE specifically, integrating automated testing directly into the modeling process may also be applicable to other conceptual modeling approaches.

7 Conclusion

This paper introduces automated Test-Driven Modeling for MERODE through MUnit, a framework comprising a DSL for MERODE test cases and an extension of the MERLIN environment to support the creation and automated execution of these tests. This tool extension replaces the previous approach, in which model validation in MERODE was performed by generating prototypes and manually testing the resulting application. The evaluation was conducted by experienced MERODE instructors, who found the DSL both usable and sufficiently expressive for testing MERODE models. Furthermore, they described integrating testing into the modeling workflow as having significant educational value, providing immediate feedback while students model. Future work will focus on evaluating MUnit in classroom settings to investigate students’ perceptions of it and its impact on learning outcomes.

Acknowledgments. We thank the participants of our study for their time and feedback. The authors declare that no AI tools were used in writing this paper and take full responsibility for its content.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Beck, K.: Test-driven development: by example. Addison-Wesley Professional (2003)
2. Cammaerts, F., Verbruggen, C., Snoeck, M.: Investigating the effectiveness of model-based testing on testing skill acquisition. In: Barn, B.S., Sandkuhl, K. (eds.) *The Practice of Enterprise Modeling*. pp. 3–17. Springer International Publishing, Cham (2022)
3. Desai, C., Janzen, D., Savage, K.: A survey of evidence for test-driven development in academia. *SIGCSE Bull.* **40**(2), 97–101 (Jun 2008). <https://doi.org/10.1145/1383602.1383644>
4. Freeman, S., Pryce, N.: *Growing object-oriented software, guided by tests*. Pearson Education (2009)
5. Martin, R.C.: *Clean code: a handbook of agile software craftsmanship*. Pearson Education (2009)
6. Mekhryukova, V., Cammaerts, F., Snoeck, M.: Investigating the effectiveness of feedback-driven exercises on deadlock detection skills in conceptual modelling. In: Paja, E., Zdravkovic, J., Kavakli, E., Stirna, J. (eds.) *The Practice of Enterprise Modeling*. pp. 140–156. Springer Nature Switzerland, Cham (2025)
7. Mellor, S., Balcer, M.: *Executable UML: A Foundation for Model-driven Architecture*. Addison-Wesley object technology series, Addison-Wesley (2002)
8. Meszaros, G.: *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series (Fowler), Pearson Education (2007)
9. Myers, G.J., Badgett, T., Thomas, T.M., Sandler, C.: *The art of software testing*, vol. 2. Wiley Online Library (2004)
10. RTI: *The economic impacts of inadequate infrastructure for software testing*. NIST Planning Report 02-3, National Institute of Standards and Technology (May 2002)
11. Sedraky, G., Snoeck, M., Poelmans, S.: Assessing the effectiveness of feedback enabled simulation in teaching conceptual modeling. *Computers & Education* **78**, 367–382 (2014). <https://doi.org/10.1016/j.compedu.2014.06.014>
12. Snoeck, M.: *Enterprise Information Systems Engineering*. The Enterprise Engineering Series, Springer Cham, 1 edn. (2014). <https://doi.org/10.1007/978-3-319-10145-3>, springer International Publishing Switzerland, Published: 01 October 2014
13. Snoeck, M.: MERLIN: An intelligent tool for creating domain models. In: Dalpiaz, F., Zdravkovic, J., Loucopoulos, P. (eds.) *Research Challenges in Information Science*. pp. 549–555. Springer International Publishing, Cham (2020)
14. Sunaric, D., Verbruggen, C., Bork, D.: *dsunaric/poem-munit-online-appendix: v1.0* (Sep 2026). <https://doi.org/10.5281/zenodo.23043264>
15. Tiukhova, E., Verbruggen, C., Baesens, B., Snoeck, M.: Learning analytics tells: Know your basics and go to class. In: Fonseca, C.M., Borbinha, J., Guizzardi, G., Aveiro, D., Liaskos, S., Keet, C.M., Serral, E., Baião, F., Araújo, J., Sales, T.P., da Silva, M.M., de Cesare, S., Pinto, H.S., Bellatreche, L., Hacks, S. (eds.) *Companion Proceedings of the 42nd International Conference on Conceptual Modeling: ER Forum, 7th SCME, Project Exhibitions, Posters and Demos, and Doctoral Consortium co-located with ER 2023, Lisbon, Portugal, November 06-09, 2023*. CEUR Workshop Proceedings, vol. 3618. CEUR-WS.org (2023), https://ceur-ws.org/Vol-3618/scme_paper_2.pdf
16. Tiukhova, E., Verbruggen, C., Laet, T.D., Baesens, B., Snoeck, M.: Learning analytics dashboard with peer comparison for student feedback in conceptual modeling education. In: Guizzardi, R.S.S., Pufahl, L., Sturm, A., van der

- Aa, H. (eds.) Enterprise, Business-Process and Information Systems Modeling - 26th International Conference, BPMDS 2025, and 30th International Conference, EMMSAD 2025, Vienna, Austria, June 16-17, 2025, Proceedings. Lecture Notes in Business Information Processing, vol. 558, pp. 301–317. Springer (2025). https://doi.org/10.1007/978-3-031-95397-2_19
17. Tort, A., Olivé, A.: First steps towards conceptual schema testing. In: CAiSE Forum (2009)
 18. Tort Pugibet, A., Olivé, A.: An approach to testing conceptual schemas. *Data & Knowledge Engineering* **69**, 598–618 (06 2010). <https://doi.org/10.1016/j.datak.2010.02.002>
 19. Verbruggen, C., Snoeck, M.: Practitioners’ experiences with model-driven engineering: a meta-review. *Softw. Syst. Model.* **22**(1), 111–129 (2023). <https://doi.org/10.1007/S10270-022-01020-1>
 20. Zhang, Y.: Test-driven modeling for model-driven development. *IEEE Software* **21**(5), 80–86 (2004). <https://doi.org/10.1109/MS.2004.1331307>
 21. Zugal, S., Haisjackl, C., Pinggera, J., Weber, B.: Empirical evaluation of test driven modeling. *International Journal of Information System Modeling and Design (IJISMD)* **4**(2), 23–43 (2013)
 22. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. In: *International Workshop on Business Process Modeling, Development and Support*. pp. 163–177. Springer (2011)
 23. Zugal, S., Pinggera, J., Weber, B.: Creating declarative process models using test driven modeling suite. In: Nurcan, S. (ed.) *IS Olympics: Information Systems in a Diverse World*. pp. 16–32. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)