Gaadha Sudheerbabu, Dragos Truscan, Mikael Manngård, Kristian Klemets

Validation of Dynamic Simulation Models using Metamorphic Testing and Given-When-Then Patterns

2025

https://doi.org/10.3384/ecp218139

Publisher's PDF

Sudheerbabu, Gaadha, et al. "Validation of Dynamic Simulation Models Using Metamorphic Testing and Given-When-Then Patterns." 2025, The 16th International Modelica&FMI Conference, September 8 – 10, 2025, Lucerne University of Applied Sciences and Arts (HSLU). , https://doi.org/10.3384/ecp218139.

Validation of Dynamic Simulation Models Using Metamorphic Testing and Given-When-Then Patterns

Gaadha Sudheerbabu[1] Dragos Truscan[1] Mikael Manngård[2] Kristian Klemets[3]

[1]Åbo Akademi University, Finland, firstname.lastname@abo.fi
[2]Novia University of Applied Sciences, Finland mikael.manngard@novia.fi
[3]Univeristy of Turku, Finland kristian.klemets@utu.fi

Abstract

As the maritime industry evolves, there is a focus on simulation-driven design, testing, and validation using novel technology solutions. Simulation models designed to represent the behaviour and features of real systems are increasingly available for testing during the early phase of the full development, but in many cases, their testing suffers from the availability of test oracles. Metamorphic testing has become increasingly used in different application domains as an approach to test systems when an explicit test oracle is unavailable. In order to increase its adoption by domain experts, we combine metamorphic testing with Behaviour Driven development for the verification and validation of simulation models. The tool-based approach facilitates automated test generation based on domain-specific custom metamorphic transformations to generate meaningful test inputs for metamorphic input relations. The method also uses features and scenarios extracted from system requirements and domain expertise to define metamorphic output relations. By automating test generation based on system behaviours as features, scenarios, metamorphic transformations, and output candidates for metamorphic relations in a Gherkin-like format, the tool enables practitioners to verify models based on domain-specific constraints and metamorphic relation checks. Our preliminary evaluation shows that the tool can detect metamorphic relations violations in the simulation models under test and that automated test generation provides improved coverage.

Keywords: Metamorphic testing, Validation, Simulation, FMI, FMU

1 Introduction

In recent times, maritime vessels have evolved into complex systems of systems with advanced automation and control systems. This evolution has made the verification and validation of maritime systems more challenging and has increased the demand for new methods for early simulation-based testing. The simulation model of a system is designed based on the properties and behaviour of the real system it represents. The *Functional Mock-Up Interface* (FMI) (Blochwitz et al. 2012) is a standard that defines a container and an interface to exchange dynamic simulation models. The container packs simulation models into an archive file called *functional mock-up unit* (FMU), which contains metadata, interface specification, binary files, data files, documentation and, optionally, the source code of the model. The goal of packing simulation models as FMUs is to exchange them between different stakeholders, such as suppliers and manufacturers, and this opens up the possibility of providing black-box models of the systems and subsystems for acceptance and integration testing. Besides the fact that, in many cases, dynamic simulation models do not come accompanied by explicit test oracles to allow automated testing, this encapsulation complicates the acceptance and integration testing processes.

Metamorphic testing (Tsong Y Chen et al. 2020) is an increasingly popular testing technique for generating test cases and verifying test results based on the properties or functional behaviour of the system. This technique also effectively addresses one of the fundamental challenges in software testing, the test oracle problem (Barr et al. 2014). The test oracle problem is the challenge of determining the expected output or behaviour of the *system under test* (SUT) for a given input. Often, when an explicit test oracle is unavailable to validate the correctness of the execution results of simulation models, they are considered non-testable (Balci 2003; Weyuker 1982).

A recent study about the research directions in metamorphic relation generation (Li et al. 2024) discusses the possibility of extending the applicability of MT in automated test generation to wider application domains that face the oracle problem. Metamorphic testing has been applied to the verification of simulation software (Murphy et al. 2011) and to the verification (Jiang et al. 2014; Lindvall et al. 2017) and validation (M. M. Olsen and M. S. Raunak 2016; M. Olsen and M. Raunak 2018) of simulation models. Recent research on the validation of simulation models using metamorphic testing shows that it has been applied to agent-based and discrete event simulation models, as well as to hybrid simulation models that include combinations of agent-based models, discrete event simulation models, and system dynamics models.

Behaviour-Driven Development (BDD) is a software development approach that extends Test-Driven Development (TDD) by focusing on the behaviour of an applica-

tion from the end user's perspective. The key idea is to write tests in a natural, readable language that describes how the application should behave under specific scenarios. BDD helps teams create a shared understanding of requirements and ensures software is aligned with business goals. BDD uses requirements patterns, such as Given-When-Then, to describe the expected behaviour of a system in a clear, consistent way. It breaks down each scenario into three logical parts: preconditions, action, and expected outcome, allowing domain experts to specify executable testing scenarios.

In this work, we propose the integration of BDD with metamorphic testing to facilitate specifying the test design by domain experts to automate the test case generation, execution, and verdict assignment for the validation of dynamic simulation models. The approach can streamline the testing process by generating the test design documentation in a practitioner-friendly BDD format and automating the different steps of the testing cycle with the following contributions:

- We propose a set of metamorphic input and output relational operators to test system dynamics simulation models.
- We provide a framework to allow domain experts to express test actions and expected behaviour in BDD format by selecting metamorphic transformations and output relations.
- We provide the tool support for automating different steps of the metamorphic testing process: the metamorphic BDD test case generation, input transformation, test execution and test verdict assignment.

The rest of the paper is divided into six sections. Section 2 describes concepts such as metamorphic testing and behaviour-driven development testing. Section 3 introduces the overview of our approach. We exemplify our approach on the simulation model of a lubricating oil cooling system in Section 4 and discuss the results in Section 5. We overview related work in Section 6, and draw the conclusions in Section 7.

2 Prerequisites

This section describes in more detail the metamorphic testing technique and behaviour-driven development testing.

2.1 Metamorphic testing

Metamorphic testing (MT) was introduced by Chen et al. (Tsong Y Chen et al. 2020) as a solution to test systems without explicit specification of the test oracle. In MT, the behavioural or functional properties of the system are defined by posing a hypothesis about using generic relations, known as *metamorphic relations (MRs)*, between different sets of inputs and their expected outputs.

The typical MT process is shown in Figure 1. A *source test case* is the first set of tests performed using *seed inputs (initial inputs)* denoted as x in Figure 1. The *seed inputs* are then transformed into *morphed inputs* denoted as x' based on a defined *metamorphic transformation*. The *follow-up test cases* are performed using the resulting *morphed inputs*.

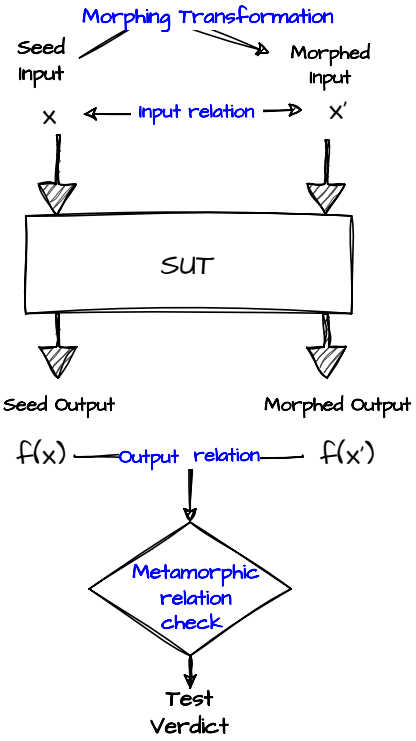

Figure 1. Generic metamorphic testing process

An MR is composed of two parts: an *input relation* and *output relation* (Liu et al. 2012). The *input relation* represents the relation between the inputs of the source test case and a follow-up test case, whereas an *output relation* represents the relation between the expected outputs of the *source and follow-up test cases*. In addition, an *implication* between the outputs of *source and follow-up test cases* is needed to specify the impact of input transformations on their corresponding outputs. The output relation, often specified using a relational operator such as equality, subset, disjoint, complete, and difference (Segura et al. 2017), should hold for any corresponding output of the system. The test verdict is assigned based on the *violation* or *non-violation of output relation*.

Several studies show that metamorphic testing has become a popular choice for testing systems without explicit test oracles and that it has been applied successfully to many application domains, including embedded systems, web applications, computer graphics, and simulation and

DOI
10.3384/ecp218139

modelling (Segura et al. 2016; Tsong Yueh Chen et al. 2018).

2.2 Behaviour Driven Development testing

Behaviour-driven development (Smart and Molak 2023) testing aims to build a development process in which development and testing activities occur concurrently throughout the product life cycle, facilitating faster and more reliable releases. The Gherkin language (Wynne and Hellesoy 2012) serves a key role in BDD and test automation by allowing developers and other stakeholders to describe software behaviour clearly that is both human-readable and executable by machines. As a structured plain-text language, Gherkin uses keywords such as *Feature, Scenario, Given, When, Then* to define the abstract description of the feature or behaviour of the system, scenarios associated with it and the steps that describe the initial conditions, test actions, and acceptance criteria. These steps ensure precise documentation of system requirements and test design as *user scenarios*. An example of a scenario specified using *Given-When-Then (GWT)* patterns is presented in Listing 1.

Listing 1. Example test specified using GWT template

```
Feature: User login
  Scenario: Successful login with valid
      credentials
    Given the user is on the login page
    When the user enters valid credentials
    Then the user should be redirected to
          the dashboard
```

3 Overview of the approach

We propose a metamorphic behaviour-driven development (Metamorphic BDD) approach for testing simulation models. The approach is intended to allow domain experts to formulate easily metamorphic tests for FMUs, by mapping concepts of MT to the GWT templates (as shown in Table 1).

Table 1. Mapping MT concepts to GWT patterns

Metamorphic concept	GWT pattern
Seed input & output	Given
Morphing Transformation Follow-up input	When
Output Metamorphic Relation	Then

Let x be an input to the SUT, and $x' = \mathrm{MT}(x)$ its metamorphic transform. The output of the SUT is $f(x)$. The metamorphic relation is evaluated by a function $\mathrm{MR}_{\mathrm{out}}(f(x), f(x'))$, which checks whether the expected relationship between the outputs holds. Consider the following GWT scenario:

$$\begin{aligned} &GIVEN \quad x \quad with \quad initial \quad value \quad x_0 \\ &WHEN \quad x' = \mathrm{MT}(x) \\ &THEN \quad \mathrm{MR}_{\mathrm{out}}(f(x), f(x')), \end{aligned}$$

In this case, all the inputs and outputs are time series over a specified time interval. As such, a metamorphic test consists of four steps:

1. *Execute source test*, with seed input x;
2. *Apply metamorphic transformation* to seed input $x' = \mathrm{MT}(x)$;
3. *Execute follow-up test* with morphed input x';
4. *Verify the metamorphic relation* between the seed output $f(x)$ and morphed output $f(x')$.

Although the approach can be applied for defining tests for any kind of simulation system, in this paper, we take advantage of the information provided in the FMU package to streamline and automate the tool support. To that extent, we use the interface specification of the SUT extracted from the FMU package and the necessary properties of the system from the requirements specification as input. This will allow us to facilitate MR definitions and to automate test generation, execution, and verdict assignment. Finally, a test report is generated as output.

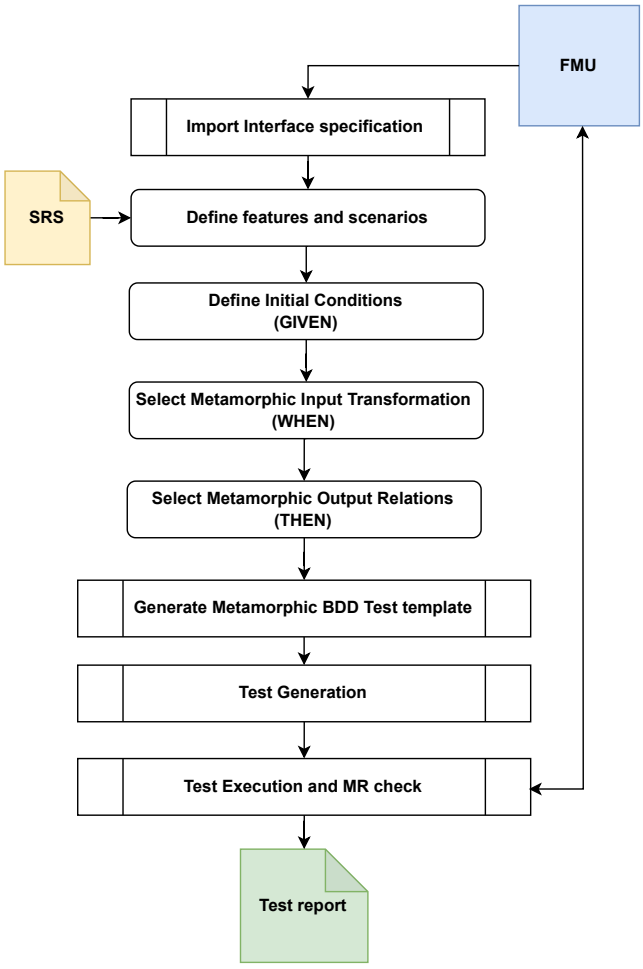

Figure 2. Metamorphic BDD testing workflow

The workflow of the approach is depicted in Figure 2. In the following, we will discuss the main steps of the approach.

Import interface specification. In the first step of our approach, the interface specification of the SUT is imported from the FMU file (*modelDescription.xml*). This includes the input and output signals, the initial values for the inputs and the ranges of the input.

Define features and scenarios. Then, the domain expert defines the features and scenarios for validating the relevant properties of the SUT.

Define initial conditions The user defines the initial condition values for the inputs of the system. These conditions correspond to the GIVEN statement of the GWT pattern.

Select Metamorphic Input Transformation. For the WHEN statement, the user selects the metamorphic transformation that should be applied to each input signal in the scenario defined. A given *metamorphic transformation* is defined as a tuple $(transformation, relation)$. The *transformation* defines the pattern based on which the input signal is changed (e.g., *None*, *Steps*, *Drift*), whereas the *relation* specifies the direction of the change (e.g., *constant*, *increase*, *decrease*). In case of *Drift* transformation, the input signals vary by a *ramp_rate* value selected from the range ± 2 and for *Steps*, they vary from the initial condition value by $\pm(value)$ until it reaches the lower and upper limits of the inputs they are applied upon. The actual value of the *ramp_rate* can be defined manually or generated using a random uniform sampling method, which is used to create the morphed input values for the corresponding input variable.

Metamorphic Output Relation Selection. The output metamorphic relation can be defined in the THEN statement, for one or several of the output signals extracted from the interface specification of the FMU. The output relations that are applicable to validate the expected properties of the output signal can be selected from the list of temporal and relational operator combinations enumerated below:

1. *Always_Greater than*: Implies that morphed outputs of the selected output variable should always be greater than the corresponding seed outputs.
2. *Always_Greater than or equal to*: Implies that morphed outputs of the selected output variable should always be greater than or equal to the corresponding seed outputs.
3. *Always_Less than*: Implies that morphed outputs of the selected output variable should always be less than the corresponding seed outputs.
4. *Always_Less than or equal to*: Implies that morphed outputs of the selected output variable should always be less than or equal to the corresponding seed outputs.
5. *Always_Equal to*: Implies that morphed outputs of the selected output variable should always be equal to the corresponding seed outputs.
6. *Eventually_Increases than*: Implies that morphed outputs of the selected output variable should eventually increase than the corresponding seed outputs.
7. *Eventually_Decreases than*: Implies that morphed outputs of the selected output variable should eventually decrease than the corresponding seed outputs.
8. *Always in given range [lower_limit upper_limit]*: Implies that morphed outputs of the selected output variable should always stay in the range defined as its lower limit and upper limit in the requirements specification.

All these metamorphic relations assume that the seed and morphed outputs will be checked at each time step of the simulation.

Metamorphic BDD test template generation. In this step, a feature file with the metamorphic BDD format is generated based on the metamorphic relations selected for the inputs and outputs in the previous step. This step takes as input the number of scenario instances to be generated and the number of time steps per scenario instance to create the parametrized BDD test file in Gherkin format.

Test Generation. In this step, the morphed test input values are generated for the follow-up tests based on the transformation relation selected and the parametrized values in the BDD test file. These test inputs are also saved to a test suite file in '.json' format.

The test inputs are generated using the *ramp_rate* added as a parametrized value in the BDD test file. The seed input is transformed into morphed input by applying the selected transformation on the initial condition value added in the GIVEN condition for the corresponding input. If the transformation selected is *None*, the value remains the same as the initial condition value throughout the simulation.

Test Execution and MR checking. In this last step, the morphed test inputs are executed on the FMU file. For this case study, the Python package `PyFMI` (Andersson, Åkesson, and Führer 2016) was used to simulate FMUs. The simulations run for the source test execution using the initial condition values for inputs and the follow-up scenario instances using the morphed input values generated. Then, each set of morphed outputs from follow-up test executions is compared with seed outputs from the source test execution, using the output MRs to assign the test verdict. The test verdict of each follow-up test execution and the comparison plots are summarized and presented in a test report for further analysis.

One should notice that each generated scenario instance would require executing a seed test and a follow-up test. However, since the seed test would be identical for each scenario instance, only one execution of the seed test is needed in order to collect the corresponding seed output. That is, if from a given scenario, *n* scenario instances are generated, the number of executions of the SUT equals *n+1*. This considerably reduces the test execution time.

DOI 10.3384/ecp218139

4 Case Study

We will exemplify the approach presented in Section 3 on a simplified version of a *Lubricating Oil Cooling* (LOC) system that transfers heat from the lubrication oil to the cooling water circuit of a ship engine unit (NoviaRDISeafaring 2024). The LOC system has a control valve which controls the lubrication oil temperature at a constant setpoint at the engine inlet using a PI controller. The controller aims to keep the lubrication oil temperature at the outlet within the boundary values specified at all conditions.

To test the LOC system using our approach, we start by importing the interface specification of the system. Then, the relevant features to be tested should be defined in the framework. For instance, a feature and scenario added to test the LOC system can be seen in Figure 3.

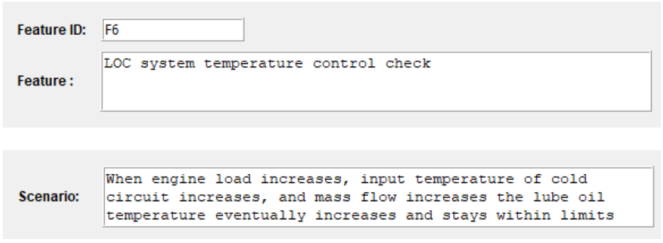

Figure 3. Defining the Feature and Scenario to test LOC system

The tool automatically extracts inputs and outputs from the model description and presents them in the GUI together with predefined data values. The initial condition values for the inputs of the system can be provided in the GIVEN tab, as in Figure 4.

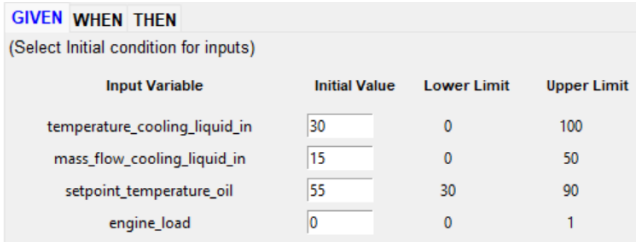

Figure 4. Setting Initial condition values

For the WHEN statement, the user selects the metamorphic transformation that should be applied for all inputs of the FMU (see Figure 5).

Figure 6 shows an example resulting input signal generated upon selecting *Steps_Increase* using a *ramp_rate* of `3.64` at a time interval of `5` for the input *temperature_cooling_liquid_in* with an initial condition value `30`. The transformation will preserve the original limits of the signal, `0` and `100`, respectively.

Figure 7 shows the output signals of the LOC system and the set of predefined output relations that can be selected for each signal. From the listed set of relations, the

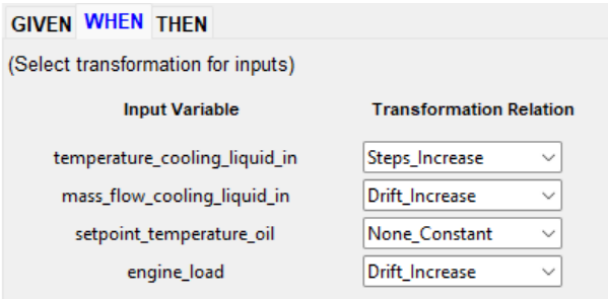

Figure 5. Metamorphic Input Transformation Selection options

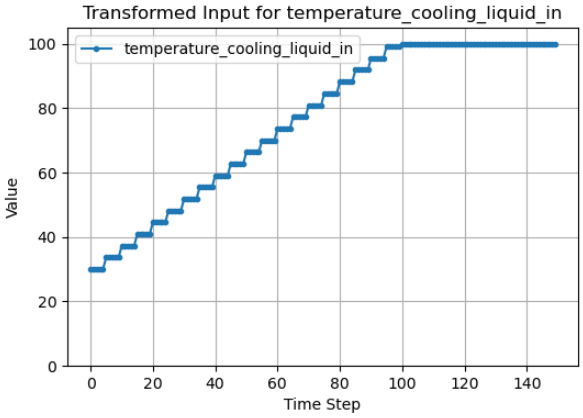

Figure 6. Example of transformed input using *Steps_Increase* transformation.

ones corresponding to the output variable selected and the input transformation applied can be chosen to facilitate the automatic generation of the metamorphic BDD test file.

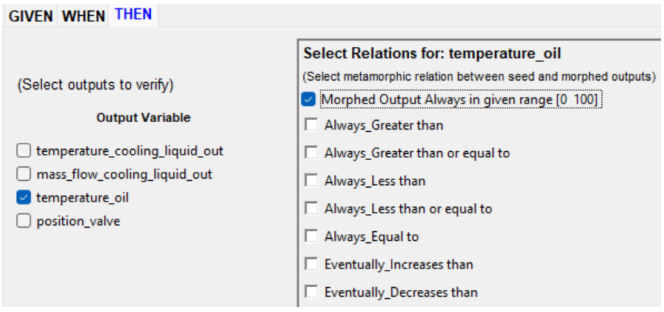

Figure 7. Metamorphic Output Relation Selection options

Figure 8 shows the first few scenario instances of the parametrized BDD test file created based on the inputs and outputs selected in Section 4, and the test generation specific parameters such as number of scenario instances to be generated as `5`, and the number of time steps per scenario instance as `1000`.

For the BDD test scenario in Figure 8, the follow-up test inputs generated for one of the scenario instances are depicted in Figure 9. Figure 10, shows the seed and mor-

```
Feature: F6_LOC system temperature control check
  Scenario: F6_scenario When engine load increases, input temperature of cold
circuit increases, and mass flow increases the lube oil temperature eventually
increases
    Given 'F6_scenario' has seed_inputs with initial values '[30, 15, 55, 0]'
generating seed_outputs in '1000' time steps

    When 'engine_load' transform by 'Drift_Increase' with ramp-rate '0.0027'
     And 'mass_flow_cooling_liquid_in' transform by 'Drift_Increase' with
ramp-rate '0.0027'
     And 'temperature_cooling_liquid_in' transform by 'Steps_Increase' with
ramp-rate '3.64'
    Then morphed_outputs of 'temperature_oil' 'Always in given range [0  100]'
     And morphed_outputs of 'position_valve' 'Always in given range [0  1]'
```

Figure 8. Metamorphic BDD test specification in GWT format

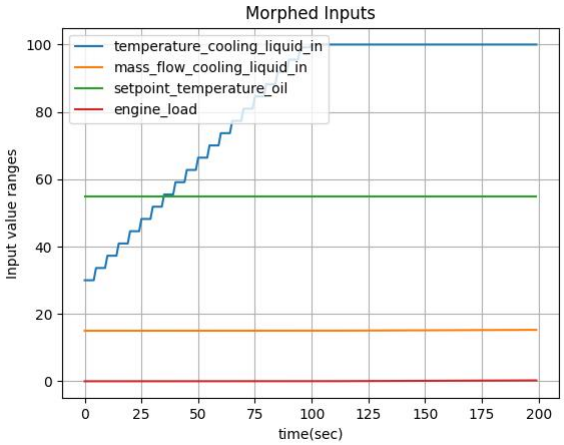

Figure 9. Plot of follow-up test generated for a scenario instance in Figure 8

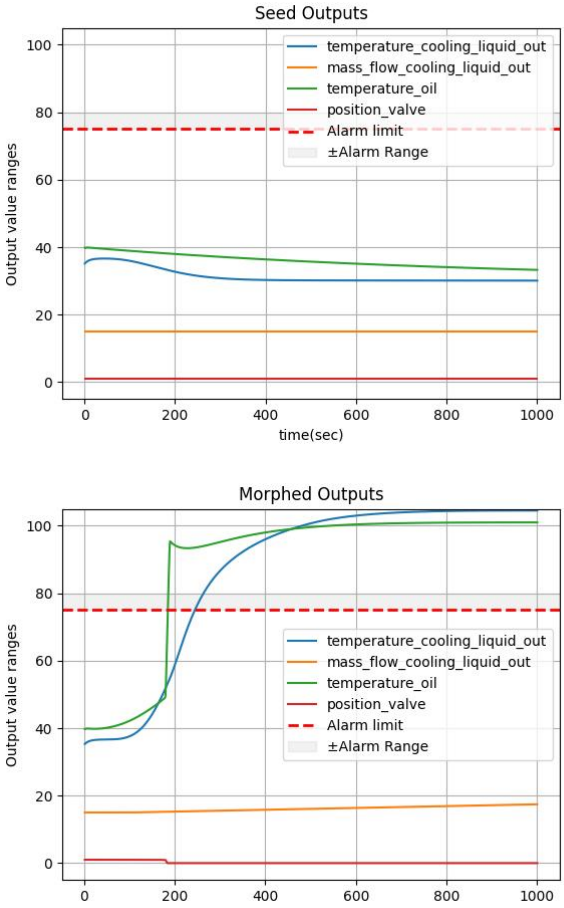

Figure 10. Example of seed and morphed output plots included in the test report

phed outputs of the scenario presented in Figure 8 where the expected behaviour of the system is that the lubricating oil temperature at the outlet eventually increases compared to the seed outputs and stays within the boundary limits in response to the input transformations depicted in Figure 9. The verdict is assigned by checking the values of *temperature_oil* from the simulation results of the source and follow-up test execution of the system using the metamorphic output relation.

4.1 Tool support

Tool support has been in Python using the GUI toolkit `Tkinter` (Lundh 1999) for the user interface (see Figure 11). Most of the steps in the metamorphic testing process are fully automated based on the input and output MR selection using the framework. To implement automation, we have used Python, Behave library, a BDD framework for Python (Behave 2012), and `PyFMI`, which interacts with the FMU generated using `Simulink` (MATLAB/Simulink 2024) to perform automated test execution. Allure (Allure 2025), an open-source tool for visualising the results of a test run, is used to generate the test report (see Figure 12), which is integrated with Behave and can also be integrated with task management tools like Jira.

5 Results and discussion

We have defined *6* features and their corresponding scenarios, and provided *1000* time steps to generate the test inputs for each execution of the SUT. The number of steps executed per feature and scenario depends on the number of scenario instances and the number of Given-When-Then templates in the BDD test file. The execution time of the tests was *1m14.584s*. The test results, comprising the features and behaviours tested, and the test verdict, are summarised in a test report in HTML format. The test suite with test inputs and outputs is saved in *'.json'* format. A domain expert performs further analysis to fix the defects identified by the failed tests, based on the details in the test report and test inputs that caused the system to fail. The failed scenario instance of *Feature 6* highlighted in Figure 12 checks if the output relation, *Always in given range [lower_limit upper_limit]*, is satisfied for the *temperature_oil*. Based on the evaluation of observed outputs, the MR is not satisfied, and the input transformations and the ramp_rates used in the scenario instance can be further analysed to fine-tune the model parameters to fix the issue.

DOI 10.3384/ecp218139

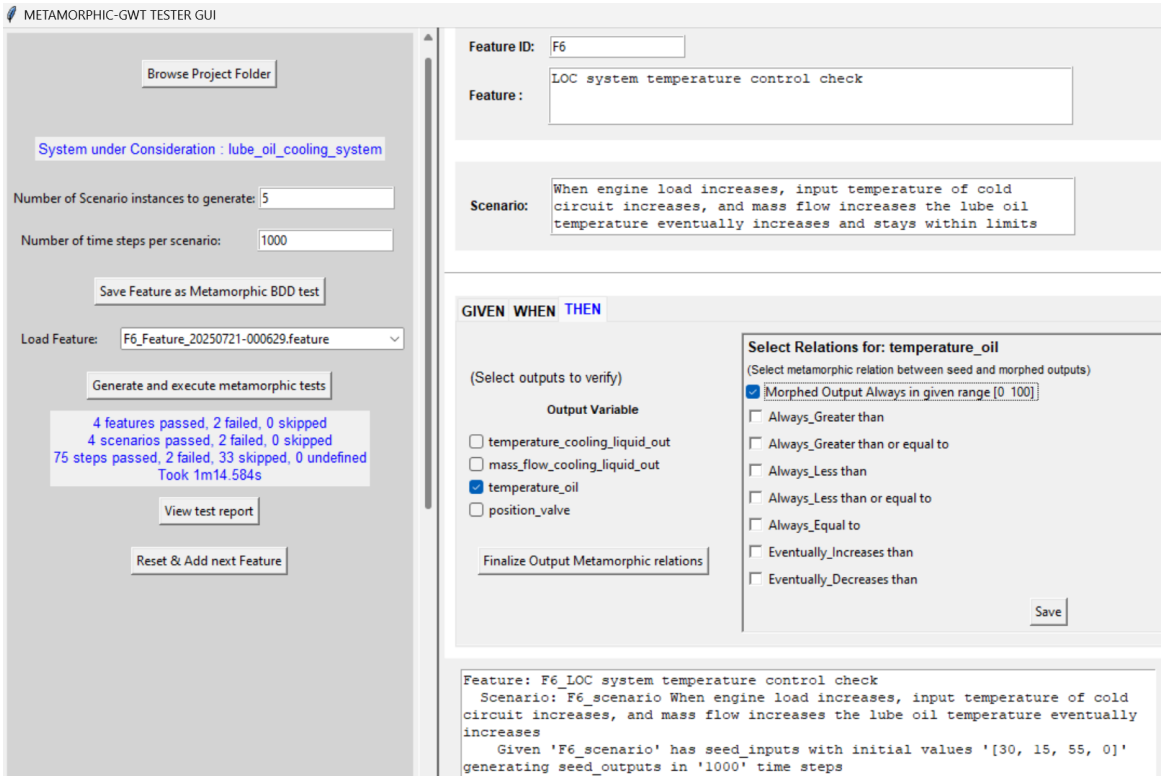

Figure 11. GUI of the Metamorphic GWT tool

6 Related work

Yao, Deng, et al. (Deng et al. 2021) proposed a declarative metamorphic testing approach, BMT, for testing autonomous driving models. The approach designed for generating MRs using custom traffic behaviours is primarily applied to image processing to verify deep learning models on their erroneous predictions. Their work also used behaviour-driven development based domain-specific language to build the metamorphic relation and to generate a test template similar to our approach. However, they target image processing systems for autonomous driving models, whereas our work focuses on testing the dynamic simulation models exported as FMUs in the maritime application domain.

7 Conclusions

We proposed a metamorphic testing approach combined with behaviour-driven development for the validation of simulation models. The main contributions of our approach were: specifying the metamorphic tests as scenarios using GWT patterns, and defining a set of metamorphic transformations and metamorphic relations to validate dynamic simulation systems. Tool support for the approach has been provided in the form of a GUI-based tool that allows the domain expert to import FMU specifications and to formulate metamorphic tests as parametrizable scenarios. The tool also allows the generation of additional tests from these scenarios. Currently, the input-output combinations in metamorphic relations are selected based on domain expertise to facilitate the test generation process. However, this can be done semi-automatically based on the availability of artefacts such as SysML diagrams, causal graphs or cause-effect diagrams (Clark et al. 2023).

Future work will focus on customizing the framework to utilize information mined from causal modelling approaches for the systematic generation of metamorphic relations. At the moment, the test generation uses random generation, but future work will replace it with a guided approach, based on our previous work on combining metamorphic testing and generative adversarial networks (Sudheerbabu et al. 2024). Future work also includes a study with practitioners on the usability and practical effectiveness of the tool to both researchers and practitioners in the context of acceptance and integration testing.

Acknowledgment

This work has been supported by Business Finland via the Virtual Sea Trial project (VST), under grant 7187/31/2023.

References

Allure (2025). *Allure report*. https://allurereport.org/docs/. Accessed: 2025-03-16.

Andersson, Christian, Johan Åkesson, and Claus Führer (2016). "Pyfmi: A python package for simulation of coupled dynamic

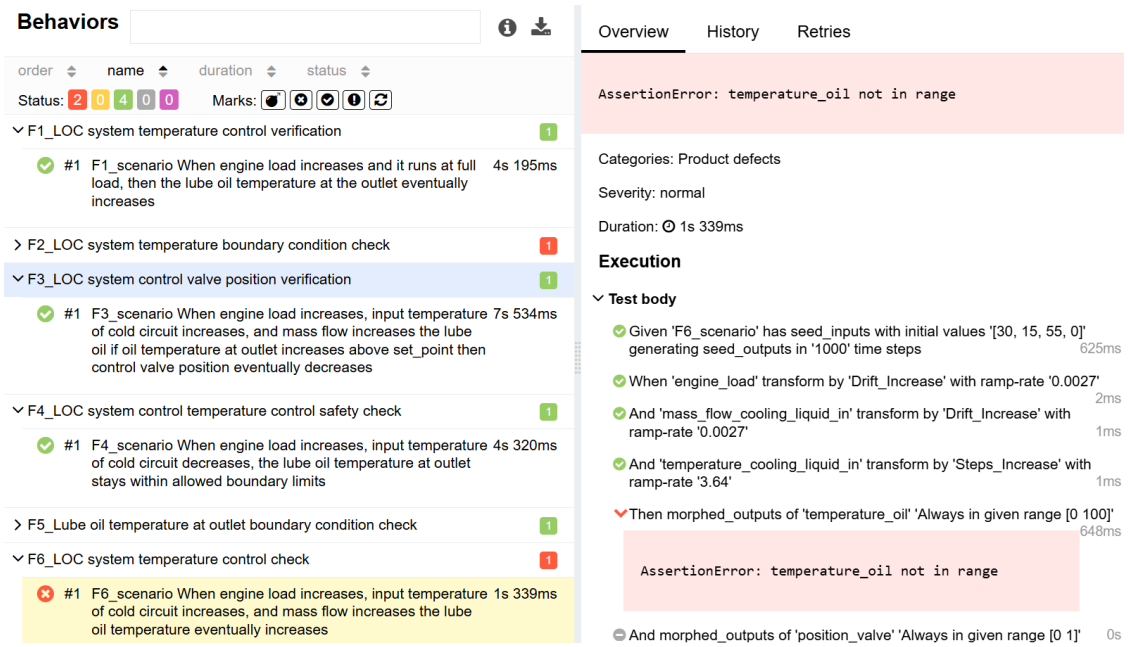

Figure 12. Test report with summary of test verdicts

models with the functional mock-up interface". In: *Technical Report in Mathematical Sciences* 2.

Balci (2003). "Verification, validation, and certification of modeling and simulation applications". In: *Proceedings of the 2003 Winter Simulation Conference, 2003.* Vol. 1. IEEE, pp. 150–158.

Barr, Earl T et al. (2014). "The oracle problem in software testing: A survey". In: *IEEE transactions on software engineering* 41.5, pp. 507–525.

Behave (2012). *Behave*. https://behave.readthedocs.io/en/latest/. Accessed: 2025-02-15.

Blochwitz, Torsten et al. (2012). "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models". In: *9th international modelica conference*. The Modelica Association, pp. 173–184.

Chen, Tsong Y et al. (2020). "Metamorphic testing: a new approach for generating next test cases". In: *arXiv preprint arXiv:2002.12543*.

Chen, Tsong Yueh et al. (2018). "Metamorphic testing: A review of challenges and opportunities". In: *ACM Computing Surveys (CSUR)* 51.1, pp. 1–27.

Clark, Andrew G et al. (2023). "Metamorphic testing with causal graphs". In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 153–164.

Deng, Yao et al. (2021). "BMT: Behavior driven development-based metamorphic testing for autonomous driving models". In: *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*. IEEE, pp. 32–36.

Jiang, Mingyue et al. (2014). "Testing model transformation programs using metamorphic testing". In.

Li, Rui et al. (2024). "Metamorphic Relation Generation: State of the Art and Research Directions". In: *ACM Transactions on Software Engineering and Methodology*.

Lindvall, Mikael et al. (2017). "Metamorphic model-based testing of autonomous systems". In: *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, pp. 35–41.

Liu, Huai et al. (2012). "A new method for constructing metamorphic relations". In: *12th Intl. Conference on Quality Software*. IEEE, pp. 59–68.

Lundh, Fredrik (1999). "An introduction to tkinter". In: *URL: www. pythonware. com/library/tkinter/introduction/index. htm* 539, p. 540.

MATLAB/Simulink (2024). *Simulink(R2024a)*. Natick, Massachusetts, United States. URL: https://www.mathworks.com.

Murphy, Christian et al. (2011). "On effective testing of health care simulation software". In: *Proceedings of the 3rd workshop on software engineering in health care*, pp. 40–47.

NoviaRDISeafaring (2024). *VST*. https://github.com/Novia-RDI-Seafaring/fmu-opc-hackathon/tree/main/fmus/loc.

Olsen, Megan and Mohammad Raunak (2018). "Increasing validity of simulation models through metamorphic testing". In: *IEEE Transactions on Reliability* 68.1, pp. 91–108.

Olsen, Megan M and Mohammad S Raunak (2016). "Metamorphic validation for agent-based simulation models." In: *SummerSim*, p. 33.

Segura, Sergio et al. (2016). "A survey on metamorphic testing". In: *IEEE Transactions on software engineering* 42.9, pp. 805–824.

Segura, Sergio et al. (2017). "Metamorphic testing of RESTful web APIs". In: *IEEE Transactions on Software Engineering* 44.11, pp. 1083–1099.

Smart, John Ferguson and Jan Molak (2023). *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster.

Sudheerbabu, Gaadha et al. (2024). "Iterative Optimization of Hyperparameter-based Metamorphic Transformations". In: *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 13–20.

Weyuker, Elaine J (1982). "On testing non-testable programs". In: *The Computer Journal* 25.4, pp. 465–470.

Wynne, Matt and Aslak Hellesoy (2012). *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf.