



Continuous Verification of Evolving Architecture Models: A Testing Framework for MontiArc

Tom Bursch
Software Engineering
RWTH Aachen University
Aachen, Germany
bursch@se-rwth.de

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
rumpe@se-rwth.de

Adrian Marin
Software Engineering
RWTH Aachen University
Aachen, Germany
marin@se-rwth.de

David Schmalzing
Software Engineering
RWTH Aachen University
Aachen, Germany
schmalzing@se-rwth.de

Abstract

Domain-specific languages and modeling languages benefit from dedicated testing frameworks that enable modelers to specify, organize, and execute tests at the same level of abstraction as the models under test. This paper presents MaUnit, a framework for specifying and executing model-level tests for MontiArc. MaUnit provides a dedicated testing notation together with an execution platform that supports reusable test components, parameterized test cases, and automated result reporting. We describe the key design decisions underlying the framework and evaluate its applicability by re-implementing the majority of MontiArc's simulator tests.

CCS Concepts

• **Computing methodologies** → **Model verification and validation**; • **Software and its engineering** → *Software testing and debugging*; Architecture description languages.

Keywords

Model Testing, ADL, Continuous Verification, Executable Models, MontiArc, Unit Testing

ACM Reference Format:

Tom Bursch, Adrian Marin, Bernhard Rumpe, and David Schmalzing. 2026. Continuous Verification of Evolving Architecture Models: A Testing Framework for MontiArc. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion 2026)*, October 04–09, 2026, Málaga, Spain. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3837062.3840373>

1 Introduction

Testing is a fundamental activity in software development [44]. It enables developers to assess software quality, detect faults early, and

validate whether an implementation satisfies the specified requirements [44]. Automated testing is particularly important in agile and iterative development processes [2], where frequent changes must be evaluated quickly and reliably. By executing test suites both locally and as part of continuous integration pipelines, developers receive rapid feedback on the impact of their modifications. In this way, automated tests provide confidence that existing functionality and previously implemented requirements remain supported as the software evolves.

For mainstream programming languages, testing frameworks provide established support [20] for creating, executing, automating, and organizing tests at the same level of abstraction as the implementation code itself. For modeling languages, and in particular for architecture description languages (ADLs), comparable language-integrated testing support is, to our knowledge, less common. This is problematic because models, like code, are engineering artifacts that evolve over time and should therefore be quality-assured and validated against their requirements [44]. Consequently, there is a need for tests of models that are reproducible, automatically executable, and developed in close integration with the models themselves, allowing both tests and models to evolve together. This is particularly relevant for executable models, where continuous and reproducible test support is necessary to validate changes early and detect regressions reliably.

In this paper, we present a framework for continuous testing of architecture models specified in the ADL MontiArc. We base its design on established testing frameworks for programming languages and adapt their core concepts to the abstraction level of architecture models. In doing so, we show how these concepts can be transferred to a modeling language and made applicable to model-based development. The contributions of this paper are twofold: first, we identify and adapt testing concepts from established programming-language testing frameworks to architecture models; second, we realize these concepts in a concrete framework for MontiArc and evaluate their applicability. The framework allows developers to define, structure, automate, and execute tests reproducibly alongside the models under test. It enables continuous validation during model evolution and supports the early detection of regressions. We present and discuss the supported testing concepts and describe the technical realization of the framework.



```

1 classdiagram Commands {
2   public enum LiftCMD {
3     UP,
4     DOWN,
5     STOP;
6   }
7   [...] // further datatypes
8 }

```

Listing 1: Datatypes describing the controls a Lift component can receive, specified as an enumeration.

The remainder of this paper is structured as follows. Section 2 introduces MontiArc, and Section 3 presents our requirements for the testing framework. Then, Section 4 provides an overview of the framework and its design decisions, and Section 5 describes its implementation. Section 6 presents the evaluation of the framework, Section 7 discusses the approach, and Section 8 compares it to related work. Finally, Section 9 concludes the paper.

2 MontiArc

MontiArc¹ [18] is a modeling language for designing and validating distributed cyber-physical system (CPS). It is grounded in the Component & Connector paradigm and has precise semantics implementing the mathematical framework Focus [5]. In Focus components are specified as stream-processing functions reacting to specific inputs, describing behavior through input-output traces. MontiArc through Focus has capabilities to model timed communication and behavior of components. The structure of MontiArc models consists of a component’s interface declaration, a component’s behavior, or a decomposed architectural model with connected subcomponents. The interface is specified through explicitly typed ports and parameters. Listing 2 depicts a MontiArc model, that describes a Lift component controlling an elevator’s movement [41]. The keyword `component` signifies a component type definition `Lift` with the port `in/out` keywords declaring directed ports. Specific timing of a port can be modeled by using the keywords `sync` or `event`. The `Lift` outputs are specified as synchronous ports, each emitting one message synchronously every time interval. The input is asynchronous, meaning the component reacts only when there is an incoming event. Datatypes in MontiArc are modeled through class diagrams [37]. Listing 1 describes the messages a `Lift` can receive on its input port.

With atomic components, component developers can describe behavior in MontiArc among other specification methods through statecharts [6]. Component developers can embed a statechart into a component using the `automaton` keyword. In asynchronous behavior, transitions fire when a stimulus is received and the guard is fulfilled. For synchronous behavior, a transition firing is conditioned by the guard being fulfilled. On firing, actions are performed on the output ports. In our example, an `UP` command in the `Wait` state sends the lift upwards, while a `STOP` command in the `Up` state stops the lift.

The `Lift` component can be instantiated in a decomposition and connected to communicating components by forwarding messages

```

1 component Lift {
2   port in LiftCMD cmd;
3   port sync out Boolean up;
4   port sync out Boolean down;
5
6   automaton {
7     initial state Wait;
8     state Up;
9
10    Wait -> Up [cmd == LiftCMD.UP] cmd;
11    Up -> Wait [cmd == LiftCMD.STOP] cmd;
12
13    Up -> Up / {
14      up = true;
15      down = false;
16    }
17    [...] // further transitions
18  }
19 }

```

Listing 2: Atomic component Lift with its behavior described by an embedded statechart. When waiting, the Lift reacts to Up commands by going up and returns to waiting if a stop command is received.

```

1 component Elevator {
2   port sync out Boolean up, down;
3   [...] // further ports
4
5   Controller ctrl;
6   Lift lift;
7   [...] // further subcomponents
8
9   ctrl.lift -> lift.cmd;
10  lift.up -> up;
11  lift.down -> down;
12  [...] // further connections
13 }

```

Listing 3: Decomposed component Elevator that describes the architecture of an elevator. A Controller sends movement signals to the Lift component, actuating the Elevator’s movements.

to their ports. Each connection has a source port and at least one target port where it forwards messages to. Listing 3 depicts an Elevator system, where the `Lift` accepts command messages from a `Controller` and forwards the produced direction messages to the outer interface of the `Elevator`.

3 Requirements for Testing Frameworks

Testing is broadly categorized into static and dynamic approaches [40]. While static testing analyzes the system under test (SUT) without execution, such as through code reviews or static analysis tools, dynamic testing requires executing the system to observe its behavior. Following this execution, test conditions derived from a test basis are evaluated to detect failures, and the

¹<https://github.com/monticore/montiarc>

results are recorded in a test protocol. This process can be performed either manually, where a tester physically interacts with the system, or automatically via dedicated test tooling. Automated testing is highly advantageous because it allows test suites to be continuously re-executed whenever changes are made. This rapid feedback loop increases both quality and productivity, enabling modern methodologies like test-driven development and shift-left testing [40].

Despite the clear advantages of test automation, developing sustainable testing practices remains a challenge across specialized domains [45]. We have identified several core areas where inadequate test tooling falls short for models with executable semantics. Primarily, many testing approaches suffer from a semantic gap, forcing modelers or domain experts to write tests using low-level implementation details rather than domain-specific constructs. This requires testers to have in-depth knowledge of the underlying general-purpose language (GPL), simulators, or third-party libraries. Creating such a high barrier to entry degrades usability and discourages the widespread adoption of testing practices [42].

Many modeling languages lack traceability and debugging support [28]. When a test fails, the resulting logs often reference deep execution traces or compiled artifacts rather than the original source specification, making it exceedingly difficult to isolate defects without deep architectural knowledge.

To overcome these limitations and avoid the common technical pitfalls that can render test automation unsustainable [12], we synthesize the following core requirements for a comprehensive testing framework based on insights from the literature and our practical experience with MontiArc.

R1: Model-Level Test Specification: Test specifications must be integrated directly into the modeling language. Testers must be able to define tests using known language constructs and concepts at the same abstraction layer as the system definition. This lowers the barrier to entry for domain experts [29, 30, 42].

R2: Automated Dynamic Execution: To manage change effectively, the framework must provide a mechanism for the automated execution of dynamic tests. Therefore, the notation must be sufficiently formal and support repeatable execution. To ensure that tests are reproducible, the framework must provide a controlled execution environment that isolates the tests from external influences [40]. This requirement is needed for modern, agile practices like test-driven development.

R3: Tests as Documentation: The test notation must be intuitive to allow the test suites to serve as clear, executable documentation of the model's behavior [46]. Consequently, boilerplate code dedicated to environment setup or resource management obscures the actual behavioral intent of the test.

R4: Model-Level Traceability: The framework must generate test protocols that summarize results at the model's abstraction level. In the event of a failure, logs must map directly back to the models, enabling users to locate defects without needing to debug generated GPL code [1].

R5: Toolchain Integration: To ensure the framework can function within modern agile processes and continuous integration pipelines,

it must be compatible with external orchestrators. It must support headless execution and output its results in standardized, machine-readable formats for pipeline gatekeeping [9, 34].

R6: Parameterization and Organization: The framework must support the parameterization of tests and provide mechanisms for organizing test suites in a way that facilitates maintenance and scalability while reducing duplicate and boilerplate code.

4 The MontiArc Test Framework

The MaUnit testing framework enables component developers to write tests for individual components or topologies of connected components. Through the testing framework, component developers can ensure that components do not only function properly in isolation but also in integration with other components. Tests are repeatable and can be executed automatically, supporting modern development workflows such as continuous integration and test-driven development.

The framework adopts concepts from the JUnit² framework and applies them to component testing. Annotations provide a structured way to declare and organize tests within the framework. Assertions verify that components behave as expected during testing by checking that the actual output of components matches the expected output. In addition to the core testing capabilities, the framework offers support for reporting and logging test results.

4.1 Writing Tests

In MaUnit, tests are identified through annotations. A component developer can mark a component definition with «test» to designate it as a test. The framework detects such annotations and thus can distinguish tests from system components. During execution, the framework collects all tests and runs them in isolation.

Both external and internal component definitions can be declared as tests. This gives component developers flexibility in how they organize tests. Developers can either create one test per file or combine multiple tests as internal component definitions. It is even possible to define tests directly as internal component definitions within the SUT. The organization of tests is ultimately up to the developer and can, for example, be defined by the team's conventions.

A test is realized by implementing the component definition declared as a test. Each test describes a complete system, *i.e.*, a fully connected topology of components without external ports. In the most general form, such tests consist of the SUT and a test oracle. The oracle represents the SUT's context: it sends inputs to the SUT and verifies its response. If the SUT does not behave as expected by the oracle, the test fails.

A test does not necessarily have to focus on a single system; it can also verify the interaction between multiple systems. Since tests are implemented as topologies of components, tests have a high degree of structural flexibility, allowing virtually any configuration to be tested. The test oracle does not have to be realized as a single component but can itself be modularized. This allows a system to be evaluated in diverse contexts using various combinations of concrete systems, mocks, and auxiliary oracles. Through such

²<https://junit.org/>

flexibility, complex system behaviors can be incrementally verified, ensuring that both individual components and their integration behave as intended. Consequently, the framework supports a broad spectrum of testing strategies, ranging from isolated unit tests to comprehensive integration tests.

```

1  component LiftTest {
2    <<test, ticks = 5>>
3    component ShouldMoveUp {
4      Lift sut;
5      sut.up -> oracle.up;
6      sut.down -> oracle.down;
7      LiftTestOracle oracle;
8      oracle.cmd -> sut.cmd;
9    }
10 }

11 component LiftTestOracle {
12   port out LiftCMD cmd;
13   port sync in boolean up, down;
14   <<delayed>> automaton {
15     initial state I;
16     state S; state R;
17     I -> S / cmd = LiftCMD.UP;
18     S -> R / assertTrue(up);
19   }
20 }

```

Listing 4: The test `ShouldMoveUp` is defined as an internal component definition of `LiftTest`, which groups multiple related tests (not depicted here). It is implemented by composing two components: the `sut`, of type `Lift`, which is the SUT, and the `oracle`, which provides the test inputs and verifies the corresponding responses of the `sut`. The type `LiftTestOracle` specifies the oracle’s behavior. For presentation purposes, it is defined here as an external component definition, although it could likewise be declared as an internal component definition of `ShouldMoveUp`.

An example of a test definition is shown in Listing 4. The annotation (l. 2) declares the inner component definition `ShouldMoveUp` (l. 3) as a test. The test is implemented as a complete system; that is, `ShouldMoveUp` itself has no ports but defines a topology of subcomponents (ll. 4–8). The test comprises two subcomponents: `sut` (l. 4), the SUT, and `oracle` (l. 7), the test oracle. The component definition `LiftTestOracle` (l. 11) defines the behavior of the test oracle. The test oracle sends the message `LiftCMD.UP` (l. 17) via port `cmd` to the SUT and subsequently expects an appropriate response via port `up` (l. 18). The answer is verified by an assertion provided by a supporting library.

As the only implementation-specific feature, the annotation exposes the configuration of the simulation steps. The parameter `ticks` specifies the number of simulation steps to be executed; in the given example, the test is executed for five simulation steps (l. 2). Specifying the number of simulation steps ensures that tests terminate, as they would otherwise continue indefinitely, potentially without activity. A test is considered successful if no error occurs during its execution, *i.e.*, within the specified number of simulation

steps. This mechanism also enables the verification of time-critical behavior.

4.2 Test Data Management

A test oracle can be used to describe complex interactions between the SUT and its context. The oracle provides input to the SUT, awaits its response, and subsequently supplies further input depending on the response received. However, developers may not always aim to test complex interactions, but rather to verify whether the SUT produces an expected output for a given input. For such cases, defining a full test oracle can be unnecessarily cumbersome. Moreover, an input-to-output relation may not be easily inferred from a state-based description. Therefore, the specification of a test oracle is unsuitable for certain test scenarios, *e.g.*, Input/Output (I/O) tests.

To simplify the process of supplying inputs to the SUT and verifying its outputs, a supporting library provides several predefined components for input generation and output verification. Specifically, the library offers two sets of components: Emitters and Asserters. Emitters, in their most general form, produce messages in a sequence defined by the developer. Asserters, not to be confused with assertion functions used within test oracles, have a single input port and verify that the received messages match an expected sequence of messages. A test fails if an Asserter receives an unexpected message or if an Asserter has not received all expected messages by the end of the test. Multiple Emitters and Asserters can be used in the same test.

Comprehensive testing can quickly lead to redundancies when similar test logic is repeatedly applied with different data. Such redundancies not only increase maintenance effort but also make it difficult to group functionally equivalent tests. Although composition, predefined library components, and reusable test case definitions help mitigate redundancies through modularization, subtle variations may still necessitate redundant test structures.

A more rigorous and scalable solution is to abstract test logic from input data and system configurations. This can be achieved, for example, through parameterized tests, which enable component configuration and variable test data. To this end, the test framework provides built-in support for parameterized testing. In such tests, the test definition includes parameters, and annotations are used to supply sets of arguments, one for each test execution. By passing these parameters to subcomponents, their behavior can be flexibly adapted while maintaining the overall test structure.

4.3 Reporting and Logging

MaUnit integrates into the existing MontiArc tooling. After a test execution, the framework automatically generates reports in various output formats within the project’s build directory. These formats include an XML file containing the results and logs of all executed tests, an HTML report designed to be opened in a web browser, and a compact binary format. The output format is fully compatible with JUnit’s output format. Hence, it integrates with standard build tools like Gradle³, appearing natively in Gradle test summaries and continuous testing environments.

³<https://gradle.org/>

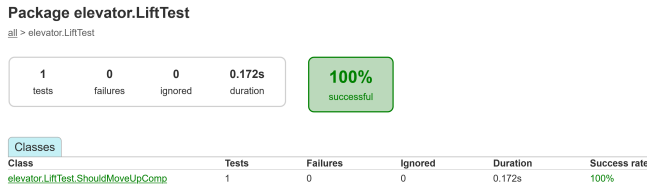


Figure 1: The HTML report generated by MaUnit for a test execution of the ShouldMoveUp test shown in Listing 4.

The generated HTML report provides a clear, high-level overview of the test execution, detailing the number of successful, failed, and ignored tests, alongside their execution duration. For example, executing a simple logic gate test suite, such as ShouldMoveUp, produces a summary page where developers can immediately see the results of all test cases as shown in Figure 1. When a test fails, the report contains a clear console message and the stack trace detailing the number of executed ticks and the execution log up to the point of failure, ensuring consistent and traceable error reporting.

4.4 Assertions and Component Libraries

To facilitate the creation of robust tests without requiring repetitive manual setups, MaUnit provides an extensive testing library. This library is split into class-based assertion methods used within behavioral definitions and reusable MontiArc components for stream generation and assertion.

Assertions. The assertion library is designed for use within the behavior definitions of atomic components and heavily mirrors standard unit testing frameworks.

- **Standard Assertions:** Modeled after the JUnit Jupiter API [17], this includes fundamental methods such as `assertTrue`, `assertFalse`, `assertEquals`, and `fail()`. Each method can accept an optional string message to provide explicit context upon failure.
- **Fluent Assertions:** For making assertions over complex objects or lists, MaUnit offers a fluent API [36]. This allows users to chain methods together, starting with `assertThat()` followed by methods such as `.hasSize(3)` and `.contains(2)` to create highly readable, sentence-like validations.

Library Components. For testing component interactions, MaUnit supplies a MontiArc component library categorized mainly into Emitters (for generating mock input streams) and Asserters (for verifying output streams).

- **AssertTrue / AssertFalse:** Simple assertion components that verify all incoming messages on an untimed port are strictly true or false, failing the test with a predefined or custom message if the condition is violated.
- **AssertEquals / AssertEqualsSync / AssertEqualsUntimed / AssertEqualsTimed:** A suite of generic components that compare actual incoming message streams against an expected value or a predefined stream. They offer varying levels of strictness and timings, allowing users to evaluate streams synchronously, untimed, or timed.

- **Emit / EmitList:** Basic emitter components that output a continuous stream of identical messages or a predefined list of messages in a single time slice.
- **EmitSync / EmitTimed:** Advanced emitters that output sequences of messages separated by ticks, providing fine-grained control over exactly when data is supplied to the SUT.

5 Implementation

In the following, we want to describe how a test definition in MontiArc is turned into a test execution. The overall architecture of the implementation is shown in Figure 2. Any test component is turned into a compiled Java class, which the Java part of MaUnit, the test engine, discovers and executes.

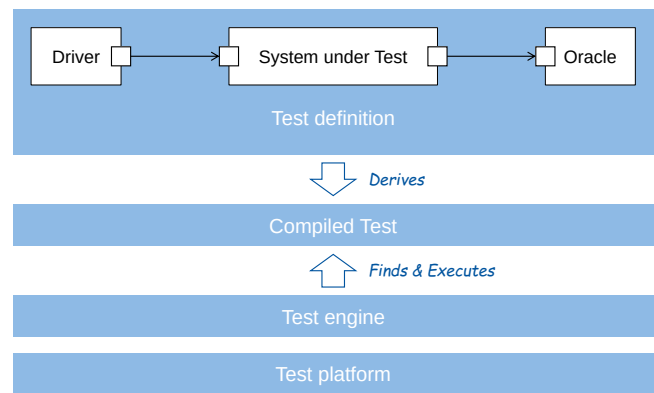


Figure 2: An abstract representation of the architecture. The test definition is transformed into a compiled test, which is then discovered and executed by the test engine.

The basis for execution is the MontiArc simulator [18]. The simulator generates Java code from MontiArc components and provides the necessary infrastructure for scheduling and running the components. MaUnit hooks into the simulation generation to provide the test specification as Java code. Here, if a component has the «test» stereotype, the generated component Java class is annotated with a `@MAUnitTest` Java annotation. Additionally, a new class is generated that implements a `MAUnitTestContext` interface, which contains all the information given by the modeler in the specification, such as the number of tests defined, the simulation length, and the parameter assignments. The context class contains generated methods for retrieving this data, and all model-level expressions are converted to Java statements.

These generated artifacts are then compiled and used by the MaUnit test engine, which is built on top of the JUnit platform [26]. Even though JUnit is most commonly known for running Java tests, it is not only a test engine but also a platform that can include different engines for different languages. Likewise, by building on this platform for MontiArc, multiple benefits arise. It provides a common understanding of tests, their results, and an API to interact with. This allows the integration of MontiArc tests with any third-party tool supporting the JUnit platform. Such tools include Gradle and IDEs for summarizing test results.

The MaUnit engine consists of two phases: a discovery phase where all tests are collected and an execution phase where every collected test is set up and simulated. During discovery, our framework finds previously generated test components by identifying classes annotated with `@MAUnitTest`. For each annotated class, it generates a container test descriptor, setting the descriptor’s name to the component’s name. This container signals to the JUnit platform that the descriptor holds a collection of child tests. These individual child test cases are only dynamically derived from the MontiArc component’s parameters in the execution stage and not included in the discovery. The descriptors are the common understanding and tell the JUnit platform and any compatible tooling how many tests there are and how to execute them. For example, some tooling could decide to only execute certain tests or display to the user information about the tests.

Following discovery, the test descriptors are executed. When the container test descriptor is asked to execute, it creates a new unique test invocation descriptor for every individual test case and parameter setup. The test invocation descriptor is then executed by first instantiating a special test scheduler to run the simulation. Unlike the default scheduler, it stores all past scheduling data. Next, the SUT, *i.e.*, the compiled test component, is instantiated, with its parameter binding resolved from the corresponding `MAUnitTestContext`. Finally, the simulation is run for the specified number of ticks.

A test is considered successful if no exception is thrown during simulation. Otherwise, the test has failed. In case of a failure, a scheduling trace is generated. The scheduling trace is given by the test scheduler; it contains the order of scheduling and all messages sent and received by components. This trace is added to the output of the test, providing the necessary information on the modeling level to identify defects.

The testing framework integrates into the MontiArc Gradle plugin, which streamlines testing by adding test tasks. Gradle’s build lifecycle and its task dependency management ensure all necessary model compilation steps are complete before the tests are run. To support a clean separation of concerns, modelers can define tests in a dedicated test source set, different from the main models. The test task can then be executed using the command line or any IDE with Gradle integration.

The provided library of components and methods is implemented using MontiArc itself. When using the Gradle plugin, a dependency on this library is added automatically. Every library component is specified using MontiArc statecharts and decomposition. The assertion methods are modeled using a class diagram and have an implementation using Java.

6 Evaluation: Testing MontiArc’s Simulator

We evaluated the presented results by reimplementing the existing MontiArc simulator tests, *i.e.*, tests that validate the functionality of MontiArc’s simulator and, in particular, its behavior with respect to the language semantics.

Verifying that a compiler preserves the semantics of its source language in the generated artifact is a non-trivial task [25, 31]. In the absence of a formal proof, tests provide a practical means of increasing confidence in a compiler’s correctness. Therefore, MontiArc provides an extensive suite of such tests, all written in

Java using JUnit. Our evaluation consisted of reimplementing these existing tests using MaUnit, enabling a direct assessment of the framework’s capabilities.

In total, there are 176 active test files containing multiple tests, each testing a specific language feature. Most of the tests follow a similar pattern: a single component is the SUT and a predefined input stream is supplied to the component. The output stream is then compared to an expected output stream. This pattern can easily be translated to MaUnit by using emitter and assertor library components combined with the parameterized test pattern.

Listing 5 partially shows an existing Java test and a semantically equivalent MaUnit test for an Inverter component. As the name suggests, the SUT, an inverter component, implements a function that inverts the incoming value. When receiving an `OnOff.ON` signal, `OnOff.OFF` is sent and vice versa. The Java test uses JUnit’s parameterized test source annotation to supply multiple input and corresponding expected output streams to the test function as lists of messages. The MontiArc test does the same, but at a much higher abstraction level. Instead of using lists to represent the streams, MontiArc supports the concept of streams and ticks, allowing the modeler to specify the streams in a more intuitive way. Similarly, the component has parameters that accept the streams and multiple test cases can be provided. These are then passed to library components that emit and assert them and are connected to the SUT. Listing 5 omits the surrounding class structure for the Java test, while the MontiArc test shows the complete component definition, with only test cases omitted for brevity.

Like the inverter test, most of the existing tests can be modeled using this pattern, and we are able to directly model 152 of the 176 test files in such a way. Semantic equivalence was established from experienced MontiArc users during construction. The results show that, on average, there are 78.67 fewer lines of code per MaUnit test file compared to the analogous Java test file, representing a reduction of 28.6%.

The remaining 24 tests we were unable to model using this pattern are deployment, scheduling, and simulation setup tests. The deployment tests use a decomposed SUT without any external ports. Since these tests hook into the subcomponent interactions and make assertions over internally sent messages, traditional MaUnit tests are not applicable, as they can only monitor the external behavior. Likewise, scheduling tests cannot be modeled in any alternative way using MaUnit. They use input streams with different tick counts, some of which are partly invalid, to assess the scheduling robustness. These input streams are not reproducible on the abstraction layer of MontiArc as any valid and simulatable component cannot produce such output streams. Simulation setup tests do not actually run the simulation but just initialize the simulator with different configurations and assert that the setup was correct. There is no adequate alternative to implementing these three types of tests with our framework, but this is arguably not needed by external users since they rarely want to test the technical implementation of the simulator itself. Even for the MontiArc development team, tests like these are rare and better implemented in Java directly. Nonetheless, as we were able to model most of the existing tests, we conclude that MaUnit is expressive enough to model a wide range of dynamic tests for MontiArc components suitable even for the MontiArc development team itself.

```

1 @ParameterizedTest
2 @MethodSource("io")
3 void testIO(List<Message<OnOff>> input,
4             List<Message<OnOff>> expected) {
5     InverterComp sut = new InverterComp();
6     PortObserver<OnOff> observer = new
7     PortObserver<>();
8     sut.port_o().connect(observer);
9     for (Message<OnOff> msg : input) {
10        sut.port_i().receive(msg);
11    }
12    sut.run();
13    Assertions.assertThat(observer)
14        .containsExactlyElementsOf(expected);
15 }
16 static Stream<Arguments> io() {
17     return Stream.of(
18         Arguments.of(
19             List.of(msg(OnOff.ON), tk()),
20             List.of(msg(OnOff.OFF), tk())
21         ),
22         [...]
23     );
24 }

```

```

1 <<test,
2 input=[
3     Sync<OnOff.OFF>,
4     [...]
5 ],
6 expected=[
7     <OnOff.ON, Tick>,
8     [...]
9 ]>>
10 component InverterTest(
11     SyncStream<OnOff> input,
12     EventStream<OnOff> expected
13 ) {
14     Inverter sut;
15
16     emitter.out -> sut.i;
17     sut.o -> assert.actual;
18
19     EmitTimed<OnOff> emitter(input);
20
21     AssertEqualsTimed<OnOff> assert(expected);
22 }

```

Listing 5: Equivalent I/O tests for an inverter component (SUT) written in Java with JUnit (Top) and MaUnit (Bottom). Both show the same test logic where an input stream of OnOff.ON is supplied, and the expected output stream is its inverted value OnOff.OFF followed by a tick. Other test cases are omitted for brevity.

The resulting tests are executed as part of the continuous integration pipeline of the MontiArc project. One benefit the MaUnit tests have over the existing Java tests is that during the evolution of the simulator, the MaUnit tests do not need to be updated as they adapt to these changes automatically.

7 Discussion

Going back to the requirements derived in Section 3, we evaluate how effectively MaUnit addresses these core needs.

By elevating the test definition to the modeling level, MaUnit successfully satisfies **R1 (Model-Level Test Specification)**. This high level of abstraction enables domain experts and non-programmers to write tests effectively [46], improving the effectiveness of modelers who are only familiar with MontiArc but no specific GPL. While not part of this work, future work can extend the framework to support testing the system-in-the-loop as well. Reusing the same test definitions and artifacts to test the final product would further strengthen the benefits of **R1**.

To address **R2 (Automated Dynamic Execution)** and **R5 (Toolchain Integration)**, MaUnit provides a fully automated execution mechanism using Gradle. Introducing any new automation framework carries the risk of technical friction [12]. To mitigate this and ensure reliability, we extensively validated MaUnit by successfully reimplementing the majority of the existing JUnit tests, confirming its results match established benchmarks. Furthermore, to satisfy **R5**, the framework outputs results in widely accepted formats, ensuring it is compatible with existing test management tools. By integrating into modern CI pipelines, MaUnit directly enables the continuous validation of models. As the system architecture grows and changes over time, this automation ensures that regressions are caught continuously without requiring manual testing efforts.

The evaluation also suggested that tests defined with MaUnit are significantly more compact and dense than their JUnit equivalents. And, because they are written in the same language as the models and focus purely on ADL concepts, the test suites are free from internal simulator implementation details. This makes the tests more suitable to serve as documentation of the system’s intended behavior that stays synchronized during model evolution, all while remaining as formally rigorous as a GPL framework (**R3 (Tests as Documentation)**).

As seen in the reporting, the framework improves traceability by enriching test outputs with domain information and simulation traces from the MontiArc models (**R4 (Model-Level Traceability)**). Failures can be mapped directly from the test log back to the model, providing testers with a clear understanding of the defect at the conceptual level.

To reduce boilerplate and reuse test architectures, parameterized tests can be employed. In combination with internal and external components and packages to manage tests, MaUnit can satisfy **R6 (Parameterization and Organization)**.

Despite these successes, MaUnit has notable limitations, meaning **R4** is currently only partially fulfilled. While log outputs are abstracted, the framework does not yet meaningfully improve the debugging process. Debugging remains a largely manual task that can require developers to inspect the generated simulator code, which breaks the otherwise high level of abstraction.

Furthermore, the abstraction layer could be improved in some areas. For example, specifying test execution length based on ticks still requires a low-level understanding of stream definitions. Future work will aim to replace this with a more abstract test duration specified in SI time units.

Finally, as models are by definition an abstraction [38], they contain inherent underspecification. In MontiArc this ranges from component scheduling to incomplete automata. The simulator must resolve these underspecifications in one concrete way, but the final target system may resolve them differently. This divergence creates a risk: a test could pass in the simulator but fail on the actual hardware (or vice versa). Therefore, to account for implementation specifics, tests must ultimately be executed on the final platform, not just within the simulator. Or all possible resolutions of underspecification must be tested.

In summary, we believe the current limitations are either addressable in future versions or are outweighed by the benefits the framework brings in usability, language independence, and automated model-level verification. The results can be generalized to other architectural modeling languages with executable semantics, as the requirements and challenges identified are not unique to MontiArc.

8 Related Work

The domain of model testing is broad. Testing models aims at showing that the behavior of a SUT differs from expected behavior and gain confidence in the system otherwise [43]. Testing frameworks such as JUnit [17], NUnit [21], and Pytest [33] provide comprehensive support for writing and executing unit tests and test suites, including configurable execution flows, extensible execution mechanisms, assertion-based validation libraries, and dedicated test runners. But employing these testing frameworks directly in the GPL used for model execution, when testing models, requires extensive knowledge of the intricacies of the modeling language tooling suite and model execution engine. MaUnit was inspired by JUnit, for one, extending the JUnit test engine platform. MaUnit integrates its concepts in the modeling language itself, improving parametrized annotations, e.g. @ValueSource and @CsvSource, extending language syntax and statically type-checking complex expressions providing language-specific support.

Using models for testing lifts tests to the problem space and provides abstraction over the actual language specific implementation of the test logic. There are approaches for test specification and test case generation with statecharts [4, 32, 38]. These approaches center around providing different coverage strategies, e.g., transition or state coverage, to create different test sets [35]. MaUnit offers state-based black-box testing for statecharts embedded in components but because MontiArc can also model decomposed architectures it is also capable of gray-box testing by providing tests for each subcomponent.

In model-driven engineering (MDE), there are approaches targeting ADL testing by transformative approaches, where the modeling language can also express properties that have to be then transformed to a formalism a verification framework must understand [8, 43]. These abstract from the formalism itself and offer automatic test case generation, but require a certification of the transformation. Another concern is the bidirectionality of the transformation to interpret results. MontiArc employs a more pragmatic approach, offering programming-oriented techniques to test the model through its simulation generated from the behavioral model. A test model can also be written in the modeling language itself,

without using complex formalisms, requiring theoretical expertise for testers.

ViTAL [10] combines EAST-ADL [3] models with UPPAAL PORT [19] timed automata to model behavior in ADLs. To simulate and verify ADLs a tester has to specify properties in Timed Computation Tree Logic (TCTL) to automatically generate test cases. One hurdle when using TCTLs for testing is their notation, which is difficult to formulate system properties in and requires a high level of expertise [14].

Gamma [15, 16] is a component-based statechart composition framework. The Gamma Tracing Language supports the description of test cases as expected behaviors of Gamma components in response to input events, together with assertions regarding the state of the component. Test cases and execution traces can be generated through an integrated model checker, and code is generated in JUnit. Gamma offers tool interoperability support by importing models directly from tool-supported modeling frameworks such as Yakindu and MagicDraw. Gamma can additionally validate liveness properties by employing model checking in addition to testing but does not yet provide simulation capabilities.

Another verification framework is HAMR [22], an AADL-based testing and formal verification framework. Modelers specify contracts inside AADL [11] models in an Assumption Guarantee formalism [24] which are then automatically tested against generated code from the ADL. HAMR automatically generates infrastructure and application skeletons for composing threads, unit tests based on contracts and deployments in C and Slang/Logika. There are also approaches in the HAMR ecosystem that integrate AADL in industry-standard systems engineering languages such as SysMLv2 as libraries [23]. Tests are executed against the SUT, not the simulation via the model. Instead of testing the deployed code against the generated contracts, MontiArc automatically generates a simulation for the modeled (decomposed) component and tests the contracts against the simulation.

Most proprietary, enterprise-grade model-based engineering (MBE) environments offer integrated testing solutions within the same modeling environment and provide dedicated test execution environments. The modeling languages used in these tools are primarily graphical and offer comprehensive user support. MathWorks Simulink [13, 27], Cameo Simulation Toolkit [7], and Sparx Enterprise Architect [39] provide modular, compositional tests, simulation, integrated testing, and test reuse. In contrast, MaUnit builds on the fact that MontiArc is a fully textual modeling language in which all features, including tests, are available in textual form. This makes MontiArc artifacts well suited as development artifacts for MDE, albeit at the expense of user-friendliness.

While most of these approaches offer more comprehensive verification suites, from formal verification via model checkers, to test case generation via formal property specifications, or user-friendly graphical tooling, MaUnit has an advantage when used day-to-day in the development process because it provides a complete, natively integrated textual approach without forcing testers to understand complex formalisms outside of the modeling language specification.

9 Conclusion

In this paper, we introduced MaUnit, a continuous testing framework tailored for MontiArc. As executable models become increasingly central to the engineering lifecycle, the necessity for robust, automated, and language-integrated testing grows. Our approach bridges the existing semantic gap by adapting proven xUnit concepts to the abstraction level of architectural models. By enabling model-level test specification, dynamic execution, and seamless integration into modern continuous integration pipelines, MaUnit empowers domain experts to validate their models without requiring deep knowledge of the underlying GPL.

The evaluation demonstrated the framework's practical applicability. By successfully re-implementing the majority of MontiArc's simulator tests, we showed that MaUnit effectively handles complex I/O stream testing while significantly reducing boilerplate code and improving test clarity. The model-level tests act as executable documentation that evolves concurrently with the system architecture, ensuring regressions are caught early and reliably in the development process. Despite these advancements, we also identified areas for further improvement. Currently, debugging failing tests still occasionally requires manual inspection of the generated code, which breaks the model-level abstraction.

Ultimately, MaUnit represents a step toward making test-driven development and continuous validation accessible in MDE. By prioritizing usability, traceability, and automation, the framework provides a sustainable foundation for assuring the quality of complex, evolving system architectures.

Acknowledgments

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Wirtschaft und Klimaschutz unter dem Förderkennzeichen 03EN3105C gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Supported by:



on the basis of a decision
by the German Bundestag

References

- [1] Mojtaba Bagherzadeh, Nicolas Hili, David Seekatz, and Juergen Dingel. 2018. MDebugger: a model-level debugger for UML-RT. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (Gothenburg, Sweden) (ICSE '18). Association for Computing Machinery, New York, NY, USA, 97–100.
- [2] Kent Beck. 2002. *Test Driven Development: By Example*. Addison-Wesley Professional.
- [3] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabo, Sandra Torchiano, Sara Tucci, et al. 2013. EAST-ADL: An architecture description language for automotive software-intensive systems. In *Embedded Computing Systems: Applications, Optimization, and Advanced Design*. IGI Global Scientific Publishing, 456–470.
- [4] Manfred Broy, Bengt Jonsson, J-P Katoen, Martin Leucker, and Alexander Pretschner. 2005. Model-based testing of reactive systems. In *Volume 3472 of Springer LNCS*. Springer.
- [5] Manfred Broy and Ketil Stølen. 2001. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer New York, New York, NY.
- [6] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Architectural Programming with MontiArcAutomaton. In *12th International Conference on Software Engineering Advances (ICSEA 2017)* (Athens, Greece). IARIA XPS Press, 213–218.
- [7] Olivier Casse. 2017. *SysML in action with Cameo systems modeler*. Elsevier.
- [8] Johan Cederbladh, Antonio Cicchetti, and Jagadish Suryadevara. 2024. Early validation and verification of system behaviour in model-based systems engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–67.
- [9] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowland, Neil A. Ernst, and Margaret-Anne Storey. 2022. Uncovering the Benefits and Challenges of Continuous Integration Practices. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2570–2583.
- [10] Eduard Paul Enoiu, Raluca Marinescu, Cristina Seceleanu, and Paul Pettersson. 2012. Vital: A verification tool for east-adl models using uppaal port. In *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. IEEE, 328–337.
- [11] Peter H Feiler and David P Gluch. 2012. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley.
- [12] Mark Fewster and Dorothy Graham. 1999. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., USA.
- [13] Federico Formica, Tony Fan, Akshay Rajhans, Vera Pantelic, Mark Lawford, and Claudio Menghi. 2023. Simulation-based testing of simulink models with test sequence and test assessment blocks. *IEEE Transactions on Software Engineering* 50, 2 (2023), 239–257.
- [14] Fernando Silvano Goncalves, David Pereira, Eduardo Tovar, and Leandro Buss Becker. 2017. Formal verification of AADL models using UPPAAL. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, 117–124.
- [15] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. 2020. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling* 19, 6 (2020), 1483–1517.
- [16] Bence Graics, Milán Mondok, Vince Molnár, and István Majzik. 2025. Model-based testing of asynchronously communicating distributed controllers using validated mappings to formal representations. *Science of Computer Programming* 242 (2025), 103265.
- [17] Shekhar Gulati and Rahul Sharma. 2017. *Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5*. Apress, Berkeley, CA.
- [18] Arne Haber. 2016. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Shaker Verlag.
- [19] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. 2008. Component-based design and analysis of embedded systems with uppaal port. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 252–257.
- [20] Paul Hamill. 2004. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly & Associates.
- [21] Bill Hamilton. 2004. *NUnit Pocket Reference: Up and Running with NUnit*. "O'Reilly Media, Inc.".
- [22] John Hatcliff, Jason Belt, Robby, and David Hardin. 2024. Integrated contract-based unit and system testing for component-based systems. In *NASA Formal Methods Symposium*. Springer, 406–426.
- [23] John Hatcliff, Jason Belt, Robby, Clint McKenzie, and Catalina Liang. 2025. End-to-end formal methods integrated development with SysMLv2 using HAMR. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 241–260.
- [24] John Hatcliff, Danielle Stewart, Jason Belt, and August Schwerdfeger. 2023. An AADL contract language supporting integrated model-and code-level verification. *ACM SIGAda Ada Letters* 42, 2 (2023), 45–54.
- [25] Tony Hoare. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM* 50, 1 (Jan. 2003), 63–69.
- [26] JUnit Team. 2025. *Test Engines :: JUnit User Guide*. Retrieved July 11, 2026 from <https://docs.junit.org/6.1.1/advanced-topics/engines.html>
- [27] Steven T Karris. 2006. *Introduction to Simulink with engineering applications*. Orchard Publications.
- [28] Faezeh Khorram, Erwan Bousse, Jean-Marie Mottu, and Gerson Sunyé. 2023. Advanced testing and debugging support for reactive executable DSLs. *Software and Systems Modeling* 22, 3 (2023), 819–845.
- [29] Tomaž Kos, Marjan Mernik, and Tomaž Kosar. 2019. A Tool Support for Model-Driven Development: An Industrial Case Study from a Measurement Domain. *Applied Sciences* 9, 21 (2019).
- [30] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifengberg, Steffen Hillemecher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. 2018. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)* (Gothenburg, Sweden). ACM, 172–180.
- [31] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115.

- [32] Chunyan Ma, Yue Li, Yunwei Dong, and Yaqi Liu. 2012. Automatic generation of systematic test cases using AADL for embedded software. *International Journal of Future Computer and Communication* 1, 2 (2012), 106.
- [33] Brian Okken. 2022. *Python testing with Pytest: simple, rapid, effective, and scalable*. The Pragmatic Programmers LLC.
- [34] Richard F. Paige and Dániel Varró. 2012. Lessons learned from building model-driven development tools. *Software & Systems Modeling* 11, 4 (01 Oct 2012), 527–539.
- [35] Sonali Pradhan, Mitrabinda Ray, and Santosh Kumar Swain. 2022. Transition coverage based test case generation from state chart diagram. *Journal of King Saud University-Computer and Information Sciences* 34, 3 (2022), 993–1002.
- [36] Ori Roth and Yossi Gil. 2023. Fluent APIs in Functional Languages. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 105 (April 2023), 26 pages.
- [37] Bernhard Rumpe. 2012. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*, 2te Auflage. Springer Berlin.
- [38] Bernhard Rumpe. 2017. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International.
- [39] Sparx Systems Pty Ltd. 2014. Enterprise Architect User Guide. <https://sparxsystems.com/bin/EAUserGuide.pdf>. Accessed: 2026-07-01.
- [40] Andreas Spillner and Tilo Linz. 2021. *Software testing foundations: A Study Guide for the Certified Tester Exam*. dpunkt. verlag.
- [41] Frank Strobl and Alexander Wisspeintner. 1999. *Specification of an Elevator Control System – An AutoFocus Case Study*. Technical Report. Technische Universität München.
- [42] Mark Utting and Bruno Legeard (Eds.). 2007. *Practical Model-Based Testing*. Morgan Kaufmann, San Francisco.
- [43] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.
- [44] Hans van Vliet. 2008. *Software Engineering: Principles and Practice* (3 ed.). Wiley.
- [45] Hui Wu, Jeff Gray, and Marjan Mernik. 2009. Unit Testing for Domain-Specific Languages. In *Domain-Specific Languages*, Walid Mohamed Taha (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–147.
- [46] Muhammad Nouman Zafar, Wasif Afzal, and Eduard Enoiu. 2021. Towards a workflow for model-based testing of embedded systems. In *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation (Athens, Greece) (A-TEST 2021)*. Association for Computing Machinery, New York, NY, USA, 33–40.