# A Unit Test Metamodel for Test Generation

Gabriel Darbord[1], Anne Etien[1], Nicolas Anquetil[1], Benoît Verhaeghe[2] and Mustapha Derras[2]

[1]*Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*
[2]*Berger-Levrault, France*

**Abstract**

Unit testing is a crucial aspect of software development, but developers often lack the time and resources to create comprehensive tests for their code. This can result in codebases that are vulnerable to bugs and issues. To address this problem, we present a unit test metamodel that enables the generation of unit tests. The metamodel provides a language-agnostic abstraction that enables automated transformation and code generation. We use the Famix family of models and tools from the Moose platform to build our metamodel and for code analysis. To generate realistic tests, we plan to use application traces consisting of method arguments and results. Our objective is to generate maintainable, human-readable tests that cover important use cases, including edge cases and rare scenarios. In this paper, we discuss related work in unit test generation, present the details of our metamodel, including its design, implementation and usage, and explore future work that will evaluate the effectiveness of our approach through case studies and experiments.

**Keywords**

unit tests, metamodel, test generation, object-oriented programming

## 1. Introduction

Unit testing is a crucial part of the software development process, enabling developers to ensure that their code works as intended and reducing the risk of introducing bugs or issues. By writing test methods for each behavior of the code under test, developers can be confident that their code will behave as expected in most situations, and that any changes they make in the future will not introduce new bugs or issues. In addition, by isolating the code under test, any issues or bugs discovered will only affect the specific unit being tested, not the entire system. This makes it easier to find and fix issues quickly, saving time and resources in the long run. However, despite the importance of unit testing, developers often lack the time to write comprehensive tests for their code, and companies may not see the value in investing in testing efforts. This can result in codebases that lack proper testing, leaving them vulnerable to bugs and issues.

To address this problem, we present a unit test metamodel that enables the generation of unit tests. The use of metamodels provides a language-, framework-, and project-agnostic abstraction that enables automated transformation and code generation. The metamodel reifies

*IWST 2023: International Workshop on Smalltalk Technologies, August 29-31, 2023, Lyon, France*
gabriel.darbord@inria.fr (G. Darbord); anne.etien@inria.fr (A. Etien); nicolas.anquetil@inria.fr (N. Anquetil); benoit.verhaeghe@berger-levrault.com (B. Verhaeghe); mustapha.derras@berger-levrault.com (M. Derras)
0000-0001-7364-7567 (G. Darbord); 0000-0003-3034-873X (A. Etien); 0000-0003-1486-8399 (N. Anquetil); 0000-0002-4588-2698 (B. Verhaeghe); 0000-0002-2227-4024 (M. Derras)

test elements as first-class citizens, meaning they can be freely used like any other object, enabling effective manipulation and transformation of test data. By building on the work done on the Famix metamodel [1] for code analysis, we can leverage its effectiveness in understanding and analyzing complex codebases. This paper represents preliminary work, we aim to further refine our approach through additional research and evaluation. To our best knowledge, this is the first work that uses such an approach for unit test generation.

To generate realistic tests, we plan to use application traces consisting of method arguments and returned values to leverage values from real business cases. Traces refer to the sequential recording of actions or operations in a system during its execution. This information is crucial as it provides an accurate depiction of how the software behaves during its run time. We will select interesting cases and build a model around them, from which we can generate unit tests that will be integrated into the codebase. This approach is a form of regression testing, which aims to identify new software bugs in existing areas of a system after changes have been made. Our objective is that this will not only save developers time but also ensure that the tests cover important use cases, including edge cases and rare scenarios.

Furthermore, we aim to generate maintainable code that is easily understood by humans. This is important because unit tests can also serve as documentation for the code they test. Human-readable and maintainable tests make it easier for developers to understand how the code works and to make changes to the codebase with confidence. Additionally, readable tests can help with onboarding new developers to a project or maintaining code that has been written by others. Ultimately, maintainable tests can reduce the amount of time spent debugging and fixing issues in the codebase.

In this paper, we begin with a discussion of related work in unit test generation. We then present the details of our unit test metamodel, including its design, implementation, and use for test generation. Finally, we discuss future work that will evaluate the effectiveness of our approach through case studies and experiments.

## 2. Related Work

In recent years, a growing number of test generation tools have been developed, resulting in different approaches to generating unit tests. Metamodels provide an abstraction of the test code that can be used to generate new test cases, and to analyze or modify existing ones using model-driven engineering techniques. However, despite the potential benefits of modeling in software testing, metamodels for representing test elements appear to be lacking in the literature. A review [2] on model-based testing techniques highlights the various uses of metamodels in software testing and engineering. While it focuses on modeling testing approaches, it also corroborates the lack of metamodels to represent unit tests.

A unit testing metamodel proposed in [3] adds test elements to a programming language metamodel. This allows for programmers to incorporate tests directly into the class being tested. This differs from the more conventional solution of writing tests separately, which we follow. In contrast, our metamodel represents unit test elements and also allows for code generation.

An approach proposed in [4] relies on the Knowledge Discovery Metamodel (KDM) as an intermediate representation of software systems and their operating environments. This is a model-driven approach to generate test cases that are compatible with the xUnit family of testing frameworks (such as JUnit or SUnit). In comparison, our approach uses a metamodel that is specific to unit tests, rather than relying on an intermediate representation such as KDM.

EvoSuite [5] is a unit test generation tool that uses evolutionary algorithms to generate new JUnit test cases, making it Java-specific. It achieves high code coverage criteria such as branch and line coverage. However, the unit tests generated by EvoSuite can differ significantly from human-written tests, making them difficult to read and understand [6]. In contrast, our metamodel is designed to be language-agnostic, and we plan to use actual traces from the system to generate tests. Furthermore, we aim to generate maintainable code that is easy for humans to understand.

The use of execution traces for software testing has been explored in several research studies, as they provide valuable insights into the behavior of a program during execution. A web testing approach presented in [7] generates test cases from user execution traces, in the absence of software models. The paper applied mutation operators to these test cases to enrich the test suite and mimic possible real-world failures. The additional tests were analyzed, and those that produced different results were retained because they exercised additional behavior of the web application under test. Such studies demonstrate the effectiveness of using execution traces for test generation, and provide insight for our own approach to using traces to identify interesting scenarios for testing.

Recently, test generation tools based on deep learning have gained a lot of attention. AthenaTest [8] is a deep learning-based tool that generates unit test cases for Java programs by learning from real-world methods and developer-written test cases. AthenaTest has been shown to outperform other unit test generation tools such as EvoSuite in terms of test coverage and readability, according to surveys of professional developers. A3Test [9] improves on AthenaTest by incorporating assertion knowledge and checking naming consistency and test signatures, leading to better results in terms of correctness and method coverage. Despite their progress, these deep learning-based tools still face challenges. One notable limitation is their inherent dependence on large amounts of training data and significant computational resources. In addition, they may not always effectively capture important edge cases, highlighting the need for human oversight and testing.

While these approaches have shown promising results in generating tests, they have some limitations, such as being specific to a particular programming language or testing framework. In this work, we propose a unit test metamodel that aims to be more generalizable and able to represent most conventional unit tests in object-oriented programming languages.

## 3. An Approach Based on Metamodels

The approach we present in this paper involves the use of metamodels to facilitate the representation and generation of unit tests. Specifically, we use three metamodels: the unit test metamodel, which represents unit test elements (shown in blue); the business metamodel (known as Famix), which represents the codebase (shown in green); and the value metamodel, which specifies the values used to test the codebase (shown in orange). In this section, we describe the unit test and value metamodels in detail, along with their entities and how they interact to produce effective unit tests. While the business metamodel is an important component of our approach, it is already established within the Moose platform and has been discussed in previous research.

## 3.1. Unit Test Metamodel

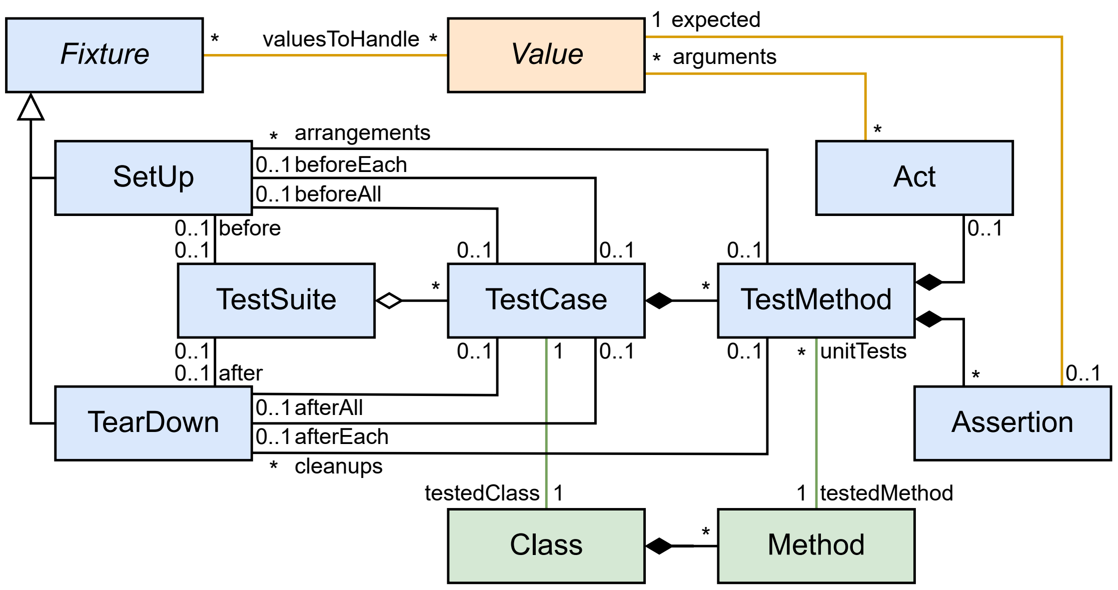

**Figure 1:** Class diagram of the Unit Test Metamodel

The *unit test metamodel*, shown in Figure 1, is a structured representation of the components that make up unit tests in object-oriented programming. It provides a way to represent, define, and generate unit tests by specifying their details, including their inputs, expected outputs, and any necessary setup or teardown steps.

Our metamodel is built around the *Arrange Act Assert* (AAA) pattern, a widely used approach to structuring unit tests. The Arrange phase involves setting up the necessary preconditions for the test, such as initializing objects and defining input values. The Act phase involves performing the method being tested, often with the input values defined in the Arrange phase. Finally, the Assert phase involves checking that the results of the Act phase are correct, typically by comparing them to expected values.

The *TestCase* entity represents a test class dedicated to testing a specific business class in the codebase. It is associated with a *Class* entity, which represents the actual class being tested. Typically, the name of the test class is derived by appending the suffix "Test" to the name of the class being tested. However, practices may vary depending on the programming language and individual project conventions. By organizing the tests for a business class into a separate test class, developers can more easily manage and maintain their unit tests, and ensure that all aspects of the business class are thoroughly tested.

A *TestMethod* is a method in a *TestCase* class that verifies a specific behavior or feature of the system under test. It is responsible for exercising the code under test and verifying its behavior.

Each test method corresponds to a *unit test*, which is a specific test that verifies the behavior of a small, isolated unit of code. A *TestMethod* is associated with a specific *Method* entity, which represents the method being tested in the codebase. Conversely, a *Method* can be associated with multiple *TestMethods*. This is because a test method uses assertions to verify that the tested method behaves as expected under different scenarios or settings.

The *Act* entity represents the “Act” phase of the AAA pattern. This phase is responsible for executing the code under test and generating the actual result. In the philosophy of unit testing, each test method should verify one and only one behavior of the system under test. Therefore, a test method should have at most one “Act” to execute the code under test and generate the expected result for that specific behavior. This ensures that each test method is focused on verifying a single behavior and simplifies the process of identifying and fixing issues that arise during testing.

An *Assertion* is a fundamental concept in unit testing. It is a condition that must be true for the test to pass, and it is used to verify that the behavior or feature being tested behaves as expected. Assertions are used to verify the results of a test by comparing expected and actual outcomes, often manifested as return values. They can also verify the correctness of data structures, network requests, user interfaces, and other system behaviors. Additionally, they allow the verification of whether a specific exception is triggered by the code under test. This can be especially useful for testing error-handling code and ensuring that the system responds appropriately to unexpected conditions.

A *TestSuite* is a higher-level entity that groups related *TestCases* together. It is a collection of test cases that share a common theme or functionality and use the same fixtures. One of the benefits of using a suite is that it allows to organize the tests in a logical and structured way. Instead of having a large number of individual test cases scattered throughout the codebase, they can be grouped into suites that make it easier to understand what is being tested and why. In addition to organizing tests, suites can also be used to manage the execution of tests. For example, there can be a suite that runs all of the tests related to a specific feature or component.

The *Fixture* is an entity used in unit testing to ensure that the system under test is executed in a consistent and isolated environment. A fixture is a set of objects and data used to configure the system before running the tests and to clean up the environment afterwards. Its subclasses, *SetUp* and *TearDown*, can be in a relationship with three different entities: *TestCase*, *TestMethod*, and *TestSuite*. When a *SetUp* is associated with a *TestCase*, it corresponds to a method that is executed before each test method or before all test methods. It is used to set up a common state or context that is shared by all the tests within the test case. When associated with a *TestMethod*, it corresponds to the code that is part of the “Arrange” phase of the AAA pattern. It is used to prepare the system under test for the specific test method being executed. When associated with a *TestSuite*, it corresponds to a method to be executed before all of the test cases contained in the suite. Similarly to the *SetUp* entity, the *TearDown* entity is associated with the same entities, but with the opposite timing and purpose. It is executed after the tests and is used to clean up any resources that were created.

Finally, the *Value* entity is used to represent runtime data in unit tests. It serves as a bridge between other entities in the system, connecting them in a logical manner. The relationship between the *Value* and *Assertion* entities represents the expected value of the assertion. When a *Value* is associated with an *Act*, it represents the arguments that will be passed to the method under test. When associated with a *Fixture*, it represents the values that will be set up or torn down. While the details of the *Value* entity will be explained in more detail later in this paper, it plays an important role in connecting the various elements of the unit test framework.

## 3.2. Value Metamodel

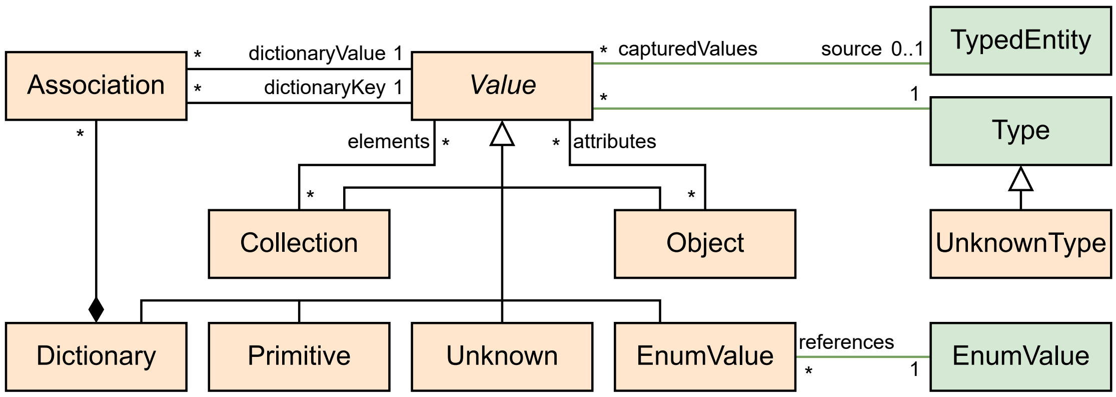

**Figure 2:** Class diagram of the Value Metamodel

The *Value metamodel*, shown in Figure 2, is a representation of runtime data in object-oriented programming languages. It is composed of different types of values, such as objects, primitives, collections, and dictionaries. The purpose of the metamodel is to provide a unified representation of runtime data that can be used for analysis, visualization, and understanding of complex object-oriented systems. By modeling the runtime data in a standardized way, the metamodel allows for the extraction of information about the structure and behavior of the system, such as the types of objects, their attributes and methods, and their relationships. An important aspect of the Value metamodel, as with the rest of our approach, is that it is designed to be language-agnostic. This is particularly useful in the context of software analysis and reverse engineering, where the code being analyzed may be written in a variety of languages. In addition, the Value metamodel offers a way of regenerating runtime data: any concrete value entity can be used to generate the code required to recreate the value it represents.

The *Value* entity serves as the root entity for all value representations in the metamodel. It is linked to a *TypedEntity*, which is a business entity that represents a concrete code element that has a type. Such a business entity can be an attribute, a parameter, or a method, among others. This link between *Value* and *TypedEntity* serves as a bridge between the code elements and the values they can hold or produce at runtime. By linking a value to its typed entity, we can retrieve the context in which the value was produced or used, and understand how it fits into the structure of the codebase.

*Value* is also in a relationship with the *Type* business entity, which is used to represent the type of the value. The reason why *Value* is related to both *TypedEntity* and *Type* is that a value may not always have a typed entity, as is the case when it is in a collection or a dictionary. However, even when a value is associated with a typed entity, its type may differ from the actual runtime type due to polymorphism. Therefore, the relationship between *Value* and *Type* is needed to accurately represent a runtime data model. In addition, the *UnknownType* entity is defined as a subclass of *Type* to be used as a placeholder when the type of a runtime value cannot be determined.

The *Object* entity represents an instance of a class. It has a relationship with *Value* to represent the values of its attributes.

The *Primitive* entity represents the value of a primitive type, such as an integer or a boolean. It stores the actual value of the data in a property. Because the data may come from an application written in a language other than the one that implements the Value metamodel, we use an equivalent primitive type in the implementing language to store the data. This allows us to store and manipulate the values of primitive types consistently and uniformly, regardless of the language in which the codebase was written.

*EnumValue* represents a member of an enumeration. The member value is represented by a reference to the corresponding enumeration element in the codebase, using the *EnumValue* business entity.

*Collection* represents a collection, such as a list or a set. It has a relationship with *Value* to represent the collection elements. Also, the same value can be in multiple collections, or in the same collection multiple times.

The *Dictionary* entity represents a collection of key-value pairs. These pairs are reified using a relationship with the *Association* entity, which in turn is related to *Value* in order to represent each pair.

Finally, the *Unknown* entity represents a value that does not have a specific type, such as the `null` value in Java. It is used in exceptional cases where the type of the value cannot be determined or is not relevant. For example, when a method returns `null` or when a variable is not initialized, it is represented as an *Unknown* entity. Note that the *Unknown* entity does not represent objects of an unknown class, as these are represented differently using a *Dictionary* of an *UnknownType*.

## 4. Unit Test Generation Using Model Transformations

In the previous section, we introduced the metamodels that we use to represent unit tests. Now we will look at the practical application of these metamodels for test generation. First, we describe how the Code and Value models are obtained and used to build Unit Test models. Then we explain how the generated test models, together with the value models, are transformed into executable code.

As shown in Figure 3, the process of obtaining a Code and Value model begins by analyzing the codebase and the traces it produces. We can then use these models to determine the elements of a unit test. In particular, we can use the trace of a particular method execution to determine the TestMethod, TestCase, and the Arrange, Act, and Assert phases of a unit test. The Result and its corresponding Method determine the TestMethod, while the Class of the Method determines the TestCase. Each Argument and Parameter determine the test Arrange and Act phases of the test, where they are set up, used and torn down. Finally, the Result obtained from the trace determines the Assert phase. We use the Result obtained from the trace as a test oracle, and the

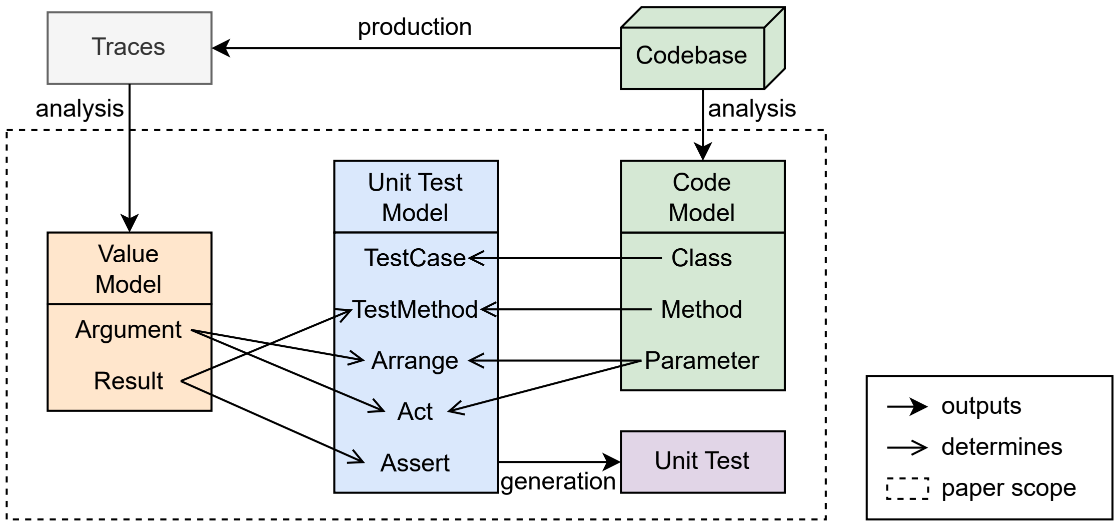

**Figure 3:** Process for deriving a unit test from value and code models

actual return value obtained in the Act phase is compared to the expected value from the trace.

Using our Unit Test and Value metamodels, we can create models that enable code generation. This process is accomplished in two steps: the first step involves exporting the model to an Abstract Syntax Tree (AST), and the second step involves visiting the AST to generate the actual code.

Exporting Unit Test and Value models to an AST involves transformation rules specific to each entity. When exporting a Value entity, the transformation rules ensure that the code necessary to recreate the represented value is generated. For collections and objects, this includes recursively exporting any dependencies of the value, such as its elements or attributes. The generated code stores the value in a variable, so that it can be accessed and manipulated in the same way as the original runtime value.

When exporting Unit Test entities, the transformation rules are more complex, except for *TestSuite* and *TestCases* which are transformed into classes without much variability. The rules for *TestMethods* orchestrate the phases of the AAA pattern within the generated methods. The transformation of *Fixtures* depends on the entities they are associated with. A fixture for a *TestSuite* or *TestCase* corresponds to a method, while a fixture for a *TestMethod* corresponds to statements. In all cases, their responsibility is to serve as a wrapper for the values they handle. An *Act* entity is transformed into the execution of the tested method with the recreated arguments, and stores the actual result in a variable. Finally, an *Assertion* generates an assert statement that verifies that the actual result matches the test oracle.

ASTs provide a structured representation of the code, allowing for easy manipulation and transformation during the code generation process. This is particularly useful for implementing optimizations and applying language-specific rules. For example, primitive literals can be inlined directly into the generated code, instead of creating and referencing a variable. By using an AST as an intermediate step, we can more easily transform the model into code in a structured and efficient manner, improving the quality and maintainability of the generated code.

## 5. Conclusion and Future Work

Unit testing is crucial in software development to ensure that code works as intended and to reduce the risk of introducing bugs, but lack of time and resources often leaves codebases vulnerable. To address this problem, we introduce a unit test metamodel that provides a language-agnostic abstraction for generating maintainable, human-readable tests. We use the Famix family of models and tools from the Moose platform for code analysis, building on existing work to create an effective and flexible solution. To complete our approach, we plan to use application traces to generate realistic tests that save developers time and ensure comprehensive coverage.

One of the major challenges in automated test generation is determining which scenarios are worth testing. Attempting to generate tests for every possible scenario is impractical because the number of possible scenarios for a given program can be prohibitively large. Furthermore, generating tests indiscriminately can result in an unwieldy and poorly organized test suite that is difficult to maintain and read, and can significantly increase the cost of software development due to the time required to execute them. Instead, we need to identify the most interesting and important scenarios that should be covered by tests. This requires analyzing the program and understanding its behavior to determine where bugs and errors are likely to occur. In addition, we need to consider factors such as reliability, coverage and maintainability when selecting test scenarios. To accomplish this, we plan to leverage application traces, which capture the sequence of method calls and their inputs and outputs during a program execution. By analyzing these traces, we can identify important use cases, edge cases, and rare scenarios that should be covered by tests.

Future work will involve evaluating the effectiveness of our approach through case studies and experiments, and exploring the ability of our metamodel to generate tests for complex and diverse codebases. We believe that our approach will make a valuable contribution to the field of unit test generation and support developers in their efforts to create robust and reliable software.

## References

[1] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, M. Derras, Modular moose: A new generation of software reengineering platform, in: International Conference on Software and Systems Reuse (ICSR'20), number 12541 in LNCS, 2020. doi:10.1007/978-3-030-64694-3_8.

[2] A. Meriem, M. Abdelaziz, A meta-model for model-based testing technique: A review, Journal of Software Engineering 12 (2018) 1–11. URL: https://scialert.net/fulltext/?doi=jse.2018.1.11. doi:10.3923/jse.2018.1.11.

[3] M. Lévesque, A metamodel of unit testing for object-oriented programming languages, arXiv preprint arXiv:0912.3583 (2009).

[4] J. Pires, F. Brito e Abreu, Knowledge discovery metamodel-based unit test cases generation, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018, pp. 432–433. doi:10.1109/ICST.2018.00056.

[5] G. Fraser, A. Arcuri, Evosuite: Automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 416–419. URL: https://doi.org/10.1145/2025113.2025179. doi:10.1145/2025113.2025179.

[6] G. Grano, S. Scalabrino, H. C. Gall, R. Oliveto, An empirical investigation on the readability of manual and generated test cases, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 348–351. URL: https://doi.org/10.1145/3196321.3196363. doi:10.1145/3196321.3196363.

[7] A. C. R. Paiva, A. Restivo, S. Almeida, Test case generation based on mutations over user execution traces, Software quality journal 28 (2020) 1173–1186.

[8] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, N. Sundaresan, Unit test case generation with transformers and focal context, CoRR abs/2009.05617 (2020). URL: https://arxiv.org/abs/2009.05617. arXiv:2009.05617.

[9] S. Alagarsamy, C. Tantithamthavorn, A. Aleti, A3test: Assertion-augmented automated test case generation, 2023. arXiv:2302.10352.